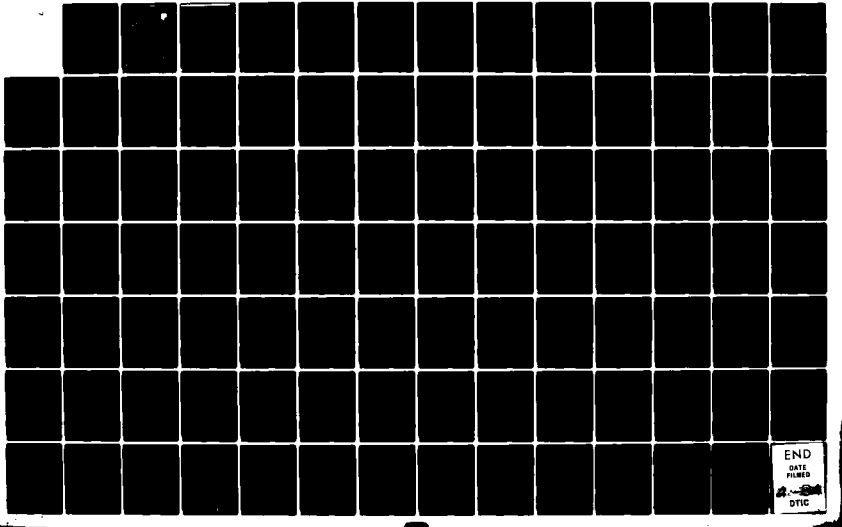
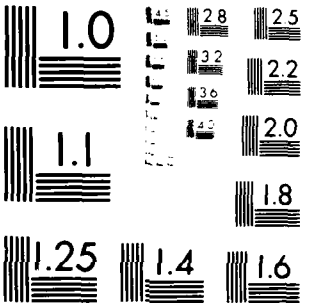
AD-A138 893 DISTRIBUTED DATABASE CONTROL AND ALLOCATION VOLUME 3 1/
DISTRIBUTED DATABASE..(U) COMPUTER CORP OF AMERICA
CAMBRIDGE MA W K LIN ET AL. OCT 83
UNCLASSIFIED RADC-TR-83-226-VOL-3 F30602-81-C-0028 F/G 9/2 NL





RADC-TR-83-226, Vol III (of three)
Final Technical Report
October 1983

DISTRIBUTED DATABASE CONTROL AND ALLOCATION Distributed Database System Designer's Handbook

Computer Corporation of America

Wente K. Lin, Philip A. Bernstein, Nathan Goodman and Jerry Nolte

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-83-226 has been reviewed and is approved for publication.

APPROVED: Emilie J. Siarkiewicz

EMILIE J. SIARKIEWICZ
Project Engineer

APPROVED:

JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER:

DONALD A. BRANTINGHAM
Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: RADC-TR-83-226, Vol III (of three)	2. GOVT ACCESSION NO.: AD-A138847
	3. RECIPIENT'S CATALOG NUMBER:
4. TITLE (and Subtitle): DISTRIBUTED DATABASE CONTROL AND ALLOCATION Distributed Database System Designer's Handbook	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report Jan 1981 - Jan 1983
	6. PERFORMING ORG. REPORT NUMBER: N/A
7. AUTHOR(s): Wente K. Lin, Philip A. Bernstein, Nathan Goodman, Jerry Nolte	8. CONTRACT OR GRANT NUMBER(s): F30602-81-C-0028
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Computer Corporation of America, Four Cambridge Center, Cambridge MA 02142	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F, 55812121
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (COTD), Griffiss AFB NY 13441	12. REPORT DATE: October 1983
	13. NUMBER OF PAGES: 98
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): Same	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Emilie J. Siarkiewicz (COTD)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Distributed Databases
Concurrency Control
Reliability

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This is the third of three volumes of the final technical report for the project "Distributed Database Control and Allocation." The first volume describes frameworks for understanding concurrency control and recovery algorithms. The second volume describes work on the performance analysis of concurrency control algorithms. This volume summarizes the results.

This volume attempts to provide a handbook of information about a number of important concurrency control algorithms which can be used in the design of

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

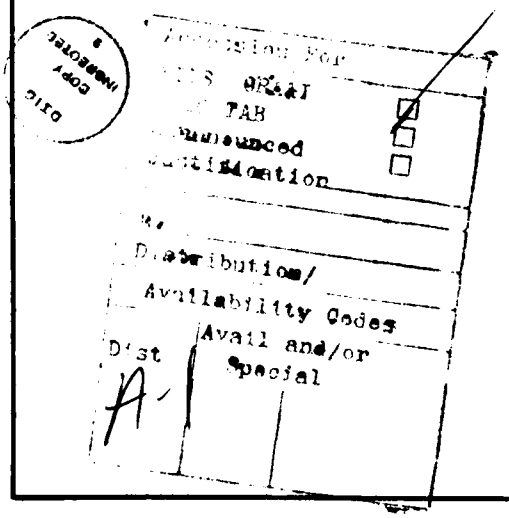
UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

a distributed ~~DBMS~~. The handbook describes a framework for distributed DBMS concurrency control which abstracts the essential structure of these algorithms from algorithmic details, and classified algorithms within this framework. The handbook then summarizes the results of a detailed simulation study of the performance of these algorithms based on the framework. For various system and application environments, algorithms are ranked according to their performance. These rankings of algorithms can guide the system designer in selecting the best distributed DBMS concurrency algorithm for his system. Additional details of the simulation results can be found in an Appendix.

In using the results, the system designer must interpret the ranking of the algorithms in the context of the performance evaluation model used in the simulation. The model either does not simulate or makes assumptions about some details of the algorithms. This is unavoidable in any simulation. However, the model used here captures all the important factors that effect the performance of a distributed concurrency control algorithm: IO delay, communication delay, CPU delay, transaction blocking through locking, transaction abortion due to conflict or deadlock, overhead for deadlock detection. Thus, the model is general enough to apply to most cases.

This handbook also provides a basis for the system designer to evaluate different database recovery algorithms. Like database concurrency control, database recovery has also been studied extensively, many algorithms have been proposed, and, on the surface, the algorithms seem very different. However, a careful examination shows that many of these algorithms are quite similar. This handbook describes a framework for database recovery algorithms. Within this framework, the many database recovery algorithms presented in the literature have been reduced to four categories. This framework can be used as the basis to compare the algorithms. Algorithms belonging to the same category may differ only in minor details.



CONTENTS

ILLUSTRATIONS

1. Introduction

The design of a distributed database management system (DBMS) involves many critical design decisions. It is recognized that one of the most important of these design decisions is the choice of the concurrency control algorithm to be used. Many concurrency control algorithms for distributed DBMSs have been proposed [Bern81a], but few studies have been undertaken to rigorously compare their performance [LIN81, LINN82a, LINN82b, LINN82c, GARC78, GARC79a, GALL82, RIES79a, RIES79b] and other characteristics. One possible reason for this is that, in detail, these algorithms seem very different, thus making comparison difficult. As a result, the distributed DBMS designer finds it difficult to choose the concurrency control algorithm which is appropriate given the design parameters of the particular system under consideration.

This report attempts to provide a handbook of information about a number of important concurrency control algorithms which can be used in the design of a distributed DBMS. The report describes a framework for distributed DBMS concurrency control which abstracts the essential structure of these algorithms from algorithmic details, and classifies algorithms within this framework. The report then summarizes the results of a detailed simulation study of the performance of these algorithms based on the framework. For various system and application environments, algorithms are ranked according to their performance. These rankings of algorithms can guide the system designer in selecting the best distributed DBMS concurrency algorithm for his system. Additional details of the simulation results can be found in an Appendix, while full details of the simulation results can be found in associated semi-annual and final technical reports [LIN81a, LIN82a, LIN82b, LIN83].

In using the results, the system designer must interpret the ranking of the algorithms in the context of the performance evaluation model used in the simulation. The model either does not simulate or makes assumptions about some details of the algorithms. This is unavoidable in any simulation. However, the model used here captures all the

important factors that effect the performance of a distributed concurrency control algorithm: IO delay, communication delay, CPU delay, transaction blocking through locking, transaction abortion due to conflict or deadlock, overhead for deadlock detection. Thus, the model is general enough to apply to most cases.

This report also provides a basis for the system designer to evaluate different database recovery algorithms. Like database concurrency control, database recovery has also been studied extensively, many algorithms have been proposed, and, on the surface, the algorithms seem very different. However, a careful examination shows that many of these algorithms are quite similar. This report describes a framework for database recovery algorithms. Within this framework, the many database recovery algorithms presented in the literature have been reduced to four categories. This framework can be used as the basis to compare the algorithms. Algorithms belonging to the same category may differ only in minor details.

The report is organized as follows. Section 2 describes the framework for distributed DBMS concurrency control algorithms, and section 3 describes the framework for database recovery algorithms. Section 4 compares the performance of various distributed DBMS concurrency control algorithms, using the framework developed in Section 2. Section 5 contains a list of references.

2. A Framework For Distributed Database Concurrency Control

2.1 Introduction

A distributed database system (DDBS) is a database system (DBS) that provides commands to read and write data that is stored at multiple sites of a network. If users access a DDBS concurrently, they may interfere with each other by attempting to read and/or write the same data. Concurrency control is the activity of preventing such behavior.

Dozens of algorithms that solve the DDBS concurrency control problem have been published (see [BERN82] and the references). Unfortunately, many of these algorithms are so complex that only an expert can understand them.

To remedy this situation, we develop, in this section, a simple framework for understanding concurrency control algorithms. The framework decomposes the problem into subproblems and gives basic techniques for solving each subproblem. To understand a published algorithm, one first identifies the technique used for each subproblem and then checks to see whether the techniques have been appropriately combined. The framework can also be used to develop new algorithms by combining existing techniques in new ways.

This section has eight subsections. Sections 2.2 and 2.3 set the stage by describing a simple DDBS architecture and sketching the framework in terms of the architecture. The framework itself appears in Sections 2.4 through 2.8. Section 2.9 uses the framework to explain several published algorithms. Section 2.10 presents a summary.

2.2 Distributed DBS Architecture

We use a simple model of DDBS structure and behavior. The model highlights those aspects of a DDBS that are important for understanding concurrency control, while hiding details that don't affect concurrency control.

A database consists of a set of data items, denoted {...,x,y,z}. In practice, a data item can be file, record, page, etc. But for the purposes of this paper, it's best to think of a data item as a simple variable. For now, assume each data item is stored at exactly one site.

Users access data items by issuing Read and Write operations. Read(x) returns the current value of x. Write(x,new value) updates the current value of x to new-value.

Users interact with the DBMS by executing programs called transactions. A transaction only interacts with the outside world by issuing Reads and Writes to the DDBS or by doing terminal I/O. We assume that every transaction is a complete and correct computation: each transaction, if executed alone on an initially consistent database, would terminate, produce correct results, and leave the database consistent.

Each site of a DDBS runs one or more of the following software modules (see Figures 2.1 and 2.2): a transaction manager (TM), a data manager (DM), or a scheduler. Transactions talk to TM's; TM's talk to schedulers; schedulers talk among themselves and also talk to DM's; DM's manage the data.

Each transaction also issues a Begin operation to its TM when it starts executing and an End when it's finished.

The TM forwards each Read and Write to a scheduler. (Which scheduler depends on the concurrency control algorithm; usually the scheduler is at the same site as the data being read or written. In some algorithms, Begins are also sent to schedulers.)

The scheduler controls the order in which DMs process Reads and Writes. When a scheduler receives a Read or Write operation, it can either outputt the operation right away (usually to a DM, sometimes to

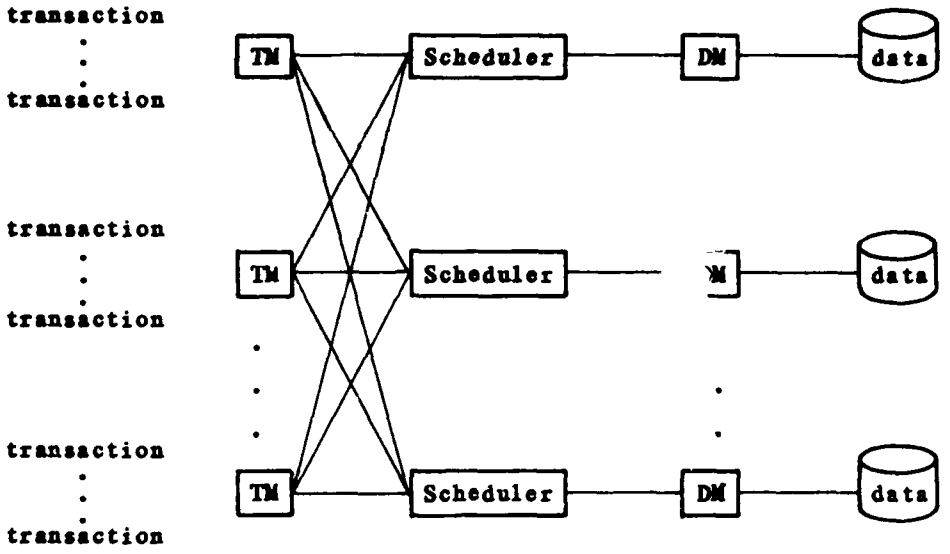


Figure 2.1 DDBS Architecture

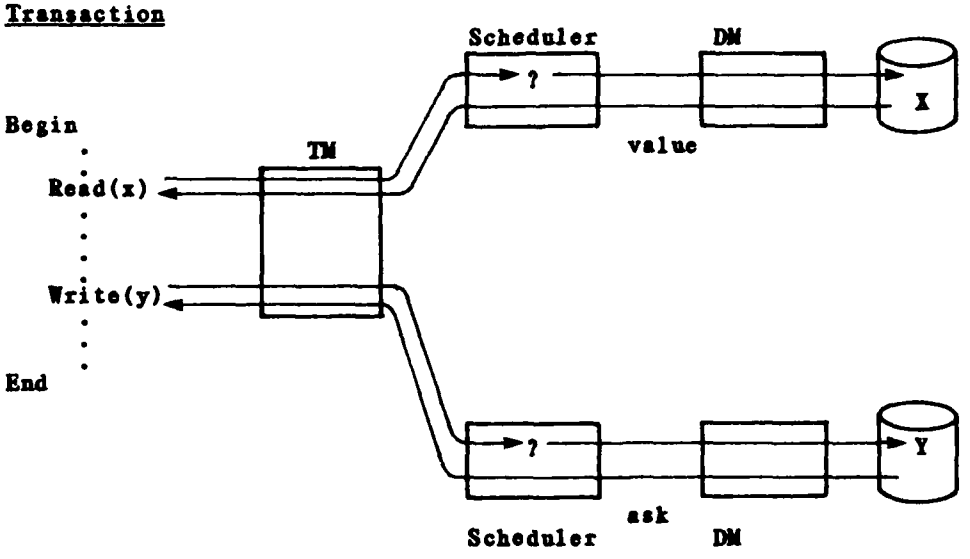


Figure 2.2 Processing Operations

another scheduler), delay the operation by holding it for later action, or reject the operation. A rejection causes the system to abort the transaction that issued the operation: every Write processed on behalf of the transaction is undone (restoring the old value of the data item), and every transaction that read a value written by the aborted transaction is also aborted. This phenomenon of one abort triggering other aborts is called cascading aborts. (It is usually avoided in commercial DBSs by not allowing a transaction to read another transaction's output until the DBS is certain that the latter transaction will not abort. In this report, we will not try to prevent cascading aborts.) Techniques for implementing abort will be discussed in Section 3. (See [GRAY81, HAMM80, LAMP76].)

The DM executes each Read and Write it receives. For Read, the DM looks in its local database and returns the requested value. For Write, the DM modifies its local database and returns an acknowledgment. The DM sends the returned value or acknowledgment to the scheduler, which relays it back to the TM, which relays it back to the transaction.

DMs do not necessarily execute operations 'first come, first served'. If a DM receives a Read(x) and a Write(x) at about the same time, the DM is free to execute these operations in either order. If the order matters (as it probably does in this case) it is the scheduler's responsibility to enforce the order. This is done by using a handshaking communication discipline between schedulers and DMs (see Figure 2.3). If the scheduler wants Read(x) to be executed before Write(x), it sends Read(x) to the DM, waits for the DM's response, and then sends Write(x). Thus the scheduler doesn't even send Write(x) to the DM until it knows Read(x) was executed. Of course, when the execution order doesn't matter, the scheduler can send operations without the handshake.

Handshaking is also used between other modules when execution order is important.

To execute Read(x) on behalf of transaction 1
followed by Write(x) on behalf of transaction 2

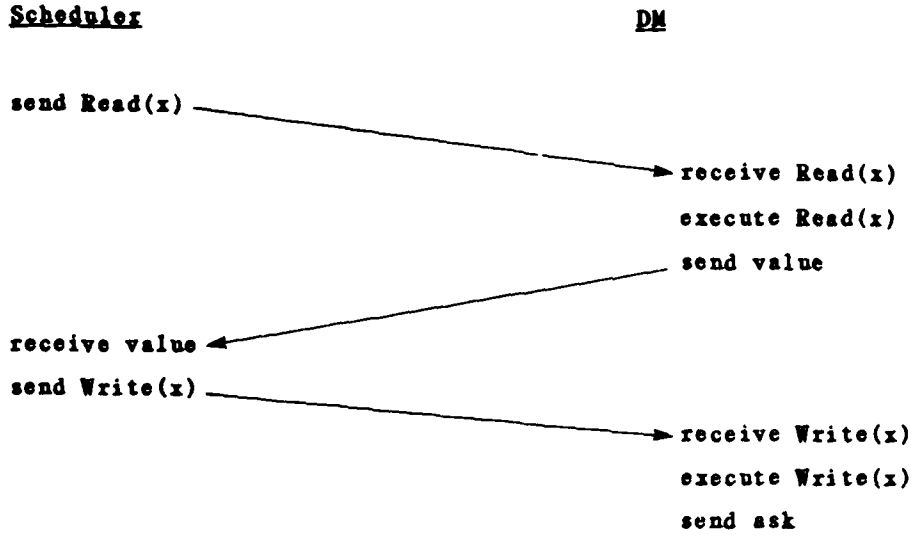


Figure 2.3 Handshaking

2.3 The Framework

The DDBS modules that are most important to concurrency control are schedulers. A concurrency control algorithm consists of some number of schedulers that run some type of scheduling algorithm in a centralized or distributed fashion. In addition, the concurrency control algorithm must handle 'replicated data'. TM's often handle this problem.

To understand a concurrency control algorithm using our framework one must determine:

1. The type of scheduling algorithm used (discussed in Sections 2.4 and 2.7)

2. The location of the scheduler(s) (i.e., centralized vs. distributed (Section 2.5))

3. How replicated data is handled (Section 2.6)

2.4 Schedulers

There are four types of schedulers: two-phase locking, timestamp ordering, serialization graph checking, and certifiers. Each type can be used to schedule rw conflicts, ww conflicts, or both. This section describes each type of scheduler and assumes that it is used for both kinds of conflict. Ways of combining scheduler types (e.g., two-phase locking for rw conflicts and timestamp ordering for ww conflicts) are described in Section 2.8. This section also assumes that the scheduler runs at a single site,(see Figure 2.4). Section 2.5 lifts this restriction.

2.4.1 Two-Phase Locking

A two-phase locking (2PL) is defined by three rules (EGLT):

1. Before outputting $r_i[x]$ (resp. $w_i[i]$), set a read-lock (resp. write-lock) for T_i on x. The lock must be held (at least) until the operation is executed by the appropriate DM. (Handshaking can be used to guarantee that locks are held long enough.)

2. Different transactions cannot simultaneously hold 'conflicting' locks. Two locks conflict if they are on the same data item and (at least) one is a write-lock. If rw and ww scheduling is done separately, the definition of 'conflict' is modified. For rw scheduling, two locks on the same data item conflict if exactly one is a write-lock (i.e., write-locks don't conflict with each other). For ww scheduling, both locks must be write locks.

3. After releasing a lock, a transaction cannot obtain any more locks.

Rule (3.) causes locks to be obtained in a two-phase manner. During its growing phase, a transaction obtains locks without releasing

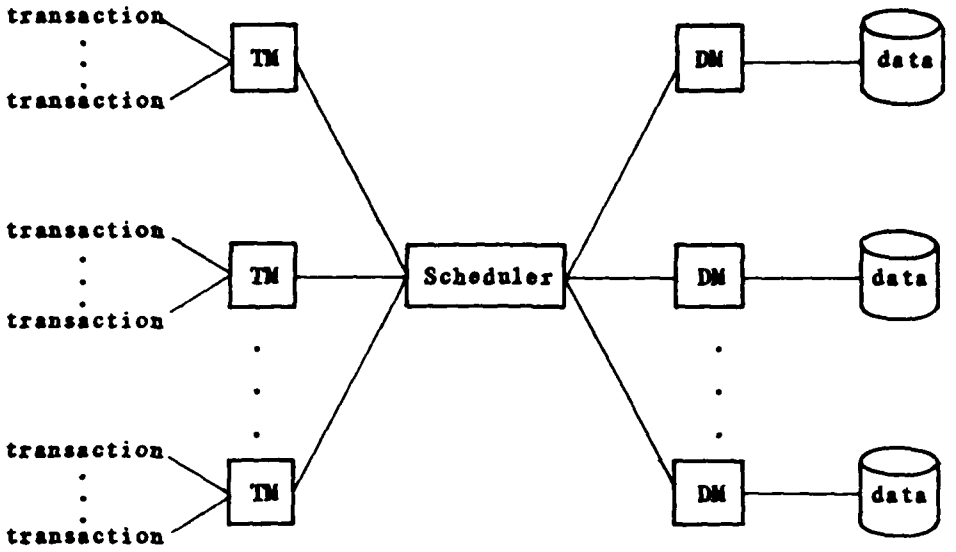


Figure 2.4 DDBS Architecture with Centralized Scheduler

any. By releasing a lock, the transaction enters its shrinking phase during which it can only release locks. Rule (3.) is usually implemented by holding all of a transaction's locks until it terminates.

Due to Rule (2.), an operation received by a scheduler may be delayed because another transaction already owns a conflicting lock. Such blocking situations can lead to deadlock. For example, suppose $r_1[x]$ and $r_2[y]$ set read-locks, and then the scheduler receives $w_1[y]$ and $w_2[x]$. The scheduler cannot set the write-lock needed by $w_1[y]$ because T_2 holds a read-lock on y. Nor can it set the write-lock for $w_2[x]$ because T_1 holds a read-lock on x. And, neither T_1 nor T_2 can release its read-lock before getting the needed write-lock because of rule (3.). Hence, we have a deadlock: T_1 is waiting for T_2 which is waiting for T_1.

Deadlocks can be characterized by a waits-for graph [HOLT72, KING74], a directed graph whose nodes represent transactions and whose edges represent waiting relationships. Edge $T_i \rightarrow T_j$ means T_i is waiting for a lock owned by T_j. A deadlock exists if and only if (iff) the

waits-for graph has a cycle. For example, in the above example the waits-for graph is

.

A popular way of handling deadlock is to maintain the waits-for graph and to periodically search it for cycles. (See [Chap. 5, AHO75] for cycle detection algorithms.) When a deadlock is detected, one of the transactions on the cycle is aborted and restarted, thereby breaking the deadlock.

2.4.2 Timestamp Ordering

In timestamp ordering (T/O) each transaction is assigned a globally unique timestamp by its TM. (See [BERN82, THOM79] for how this is done.) The TM attaches the timestamp to all operations issued by a transaction. A T/O scheduler is defined by a single rule: Output all pairs of conflicting operations in timestamp order. Make sure conflicting operations are executed by DMs in the order they were output. (Handshaking can be used to make sure of this.) As for 2PL, the definition of 'conflicting operation' is modified if rw and ww scheduling are done separately.

Several varieties of T/O schedulers have been proposed. We only sketch these variations here. Full details appear in [BERN82].

A basic T/O scheduler outputs operations in essentially first come, first served order, as long as the T/O scheduling rule holds. When the scheduler receives $r_i[x]$ it does the following:

if TS(i) < largest timestamp of any Write on x yet 'accepted'
then reject $r_i[x]$
else 'accept' $r_i[x]$ and output it as soon as all Writes on x with smaller timestamp have been acknowledged by the DM.

When the scheduler receives $w_i[y]$ it behaves as follows.

if TS(i) < largest timestamp of any Read or Write on x yet 'accepted'
then reject $w_i[x]$
else 'accept' $w_i[x]$ and output it as soon as all Reads and Writes on x with smaller timestamp have been acknowledged by the DM.

A conservative T/O scheduler avoids rejections by delaying operations instead. An operation is delayed until the scheduler is sure that outputting it will cause no future operations to be rejected. Conservative T/O requires that each scheduler receive Reads and Writes from each TM in timestamp order. To output any operation, the scheduler must have an operation from each TM in its 'input queue'. The scheduler then 'accepts' the operation that has the smallest timestamp. 'Accept' means to remove the operation from the input queue and to output it as soon as all conflicting operations that have smaller timestamp have been acknowledged by the DM. Variations on conservative T/O are discussed in [BERN82, BERN80a, LIN79].

Basic T/O and conservative T/O are endpoints of a spectrum. Basic T/O delays operations very little, but it tends to reject many operations. Conservative T/O never rejects operations, but it tends to delay them often. One can imagine T/O schedulers between these extremes. To our knowledge, no one has yet proposed such a scheduler.

Thomas' write rule (TWR) is a technique that reduces delay and rejection [THOM79]. TWR can be used only to schedule Writes, and it needs to be combined with basic or conservative T/O to yield a complete scheduler. If we're interested only in ww scheduling, TWR is simple. When the scheduler receives $w_i[y]$ it does the following:

```
if   TS(i) < largest timestamp of any Write on x yet 'accepted'
then 'pretend' to execute w_i[y] (i.e., send an acknowledgement back
       to the TM, but don't send the Write to the DM
else 'accept' w_i[x] and process it as usual.
```

The basic T/O-TRW combination works like this. Reads are processed exactly as in the basic T/O. But when the scheduler receives a $w_1[y]$, it combines the basic T/O rule with TWR as follows:

```
if   TS(i) < largest timestamp of any Read
       on x yet 'accepted'   rw scheduling (basic T/O)
then 'reject' w_i[y]
elseif TS(i) < largest timestamp of any Write on x yet 'accepted'
       ww scheduling (TWR)
then 'pretend' to execute w_i[y]
else 'accept' w_i[x] and output it as soon as all operations on x with
```

smaller timestamp have been acknowledged by the DM.

The conservative T/O-TWR combination is described in [BERN82].

2.4.3 Serialization Graph Checking

A serialization graph (SG) is a directed graph whose nodes are transactions) such as T_0, ..., T_n -- and whose edges are all $T_i \rightarrow T_j$ such that, for some x, either (1.) T_i reads x before T_j writes x, or (2.) T_i writes x before T_j reads x, or (3.) T_i writes x before T_j writes x. A serialization graph checking scheduler works by explicitly building a serialization graph (SG) and checking it for cycles. Like basic T/O, an SG checking scheduler never delays an operation (except for handshaking reasons). Rejection is the only action used to avoid incorrect execution.

An SG checking scheduler is defined by the following rules.

1. When transaction T_i Begins, add node T_i to SG.
2. When a Read or Write from T_i is received, add all edges $T_i \rightarrow T_j$ such that T_j is a node of SG, and the scheduler has already output a conflicting operation from T_j. As for the previous schedulers, the definition of 'conflicting operation' is modified if rw and ww conflicts are scheduled separately.
3. If after Rule 2 SG is still acyclic, output the operation. Make sure that conflicting operations are executed by DMs in the order they were output. (Handshaking can be used for this.)
4. If after Rule 2 SG has become cyclic, reject the operation. Delete node T_i from all edges $T_i \rightarrow T_j$ or $T_j \rightarrow T_i$ from SG. (SG is now acyclic again.)

One technical problem with SG checkers is that a transaction must remain in SG even after it has terminated. A transaction can be deleted from SG only when it is a source node of the graph (i.e., when it has no incoming edges). See [CASA79] for a discussion of this problem and for techniques that efficiently encode information about terminated transactions that remain in SG.

2.4.4 Certifiers

The term certifier refers to a scheduling philosophy, not a specific scheduling rule. A *certifier* is a scheduler that makes its decisions on a per-transaction basis. When a certifier receives an operation, it internally stores information about the operation and outputs it as soon as all earlier conflicting operations have been acknowledged. When a transactions ends, its TM sends the End operation and outputs it as soon as all earlier conflicting operations have been acknowledged. At this point, the certifier checks its stored information to see whether the transaction executed serializably. If it did, the certifier *certifies* the transaction, allowing it to terminate; otherwise, the certifier aborts the transaction.

All of the earlier schedulers can be adapted to work as certifiers. Here is an SG checking certifier. When a certifier receives an operation, it adds a node and some edges to SG as explained in the previous section. The certifier does not check for cycles at this time. When a transaction, T_i, ends, the certifier checks SG for cycles. If T_i does not lie on a cycle, it is certified; otherwise it is aborted.

Here is a 2PL certifier [THOM79, KUNG79]. Define a transaction to be *active* from the time the certifier receives its first operation until the certifier processes its End. The certifier stores two sets for each active transaction T_i:

T_i's *readset*, RS(i) = {x|the certifier has output $r_i[x]$}

T_i's *writeset*, WS(i) = {x|the certifier has output $w_i[x]$}.

The certifier updates these sets as it receives operations. When the certifier receives End_i, it runs the following test:

Let RS(active) = U(RS(j), such T_j is active, but $j \neq i$)
WS(active) = U(WS(j), such T_j is active, but $j \neq i$)
if RS(i) ∩ WS(active) ≠ ∅, or
WS(i) ∩ RS(active) ∪ WS(active) ≠ ∅
then certify T_i
else abort T_i.

This amounts to pretending that transactions hold imaginary locks on their readsets and writesets. When transaction T_i ends, the certifier sees whether T_i's imaginary locks conflict with the imaginary locks

held by other active transactions. *If there is no conflict,* T_i *is certified.* Otherwise i.. is aborted.

T/O certifiers are also possible. To our knowledge, no one has proposed this algorithm yet. Certifiers also can be built that will check for serializable executions during transactions' executions, not just at the end. The extreme version of this idea is to check for serializability on every operation. At this extreme, the certifier reduces to a 'normal' scheduler.

2.5 Scheduler Location

The schedulers of Section 2.5 can be modified to work in a distributed manner. Instead of one scheduler for the whole system, we now assume one scheduler per DM (refer back to Figure 2.1). The scheduler normally runs at the same site as the DM and schedules all operations that the DM executes.

The new issue in this setting is that the distributed schedulers must cooperate to attain the scheduling rules of Section 2.5.

The main problem caused by distributed schedulers is the maintenance of global data structures. Distributed 2PL schedulers need a global waits-for graph. Distributed SG checkers need a global SG. In distributed T/O scheduling, no global data structures are needed; each scheduler can make its scheduling decisions using local copies of R-TS(x) and W-TS(x) for each x at its DM. Distributed certifiers generally manifest the same problems as their corresponding schedulers.

2.5.1 Distributed Two-Phase Locking

Refer to the 2PL scheduling rules of Section 2.5.1. Rules (1.) and (2.) are 'local'. The scheduler for data item x schedules all cooperations on x. Hence this scheduler can set all locks on x. Rule (3.) requires a small amount of inter-scheduler cooperation, no scheduler can obtain a lock for transaction T_i after any scheduler releases a lock for T_i. This can be done by handshaking between TM's and schedulers. When T_i Ends, its TM waits until all of T_i's Reads and Writes are

acknowledged. At this point the TM knows that all of T_i's locks are set and that it's safe to release locks. The TM forwards End_i to the schedulers, which then release T_i's locks.

One problem with distributed 2PL is that multi-site deadlocks are possible. Suppose x and y are stored at sites A and B, respectively. Suppose that $r_i[y]$ is processed at A, setting a read-lock on x for T_i at A; and suppose that $r_j[x]$ is processed at site B, setting a read-lock on y for T_j at B. If $w_j[x]$ and $w_i[y]$ are now issued, a deadlock will result; T_j will be waiting for T_i to release its lock on x at A, and T_i will be waiting for T_j to release its lock on y at B. Unfortunately, the deadlock isn't apparent by looking at site A or site B alone. Only when taking the union of the waits-for graph at both sites does the deadlock cycle materialize.

See [MENA79, STON79, GLIG80, ROSE78] for solutions to this problem.

2.5.2 Distributed Timestamp Ordering

T/O schedulers are easy to distribute because the T/O scheduling rule of Section 2.5.2 is inherently local. Consider a basic T/O scheduler for data item x. To process an operation on x, the scheduler needs to know only whether a conflicting operation that has a larger timestamp has been accepted. Since the scheduler handles all operations on x, it can make this decision itself.

2.5.3 Distributed Serialization Graph Checking

SG checkers are harder to distribute than the other schedulers because the serialization graph (SG) is inherently global. A transaction that accesses data at a single site can become involved in a cycle that spans many sites. See [CASA79] for a discussion of this problem.

2.5.4 Distributed Certifiers

Distributed certifiers have a synchronization requirement a bit like Rule (3.) of 2PL: T_i's TM must not send End_i to any certifier until all of T_i's Reads and Writes have been acknowledged. (i.e., we must not try to certify T_i at any site until we are ready to certify T_i at all sites).

Beyond this, each distributed certifier behaves like the corresponding scheduler. A distributed 2PL certifier needs little inter-scheduler cooperation (beyond that described in the previous paragraph). The certifier at each site keeps track of the data items *at its site* read or written by active transactions. When a certifier at site A receives End_i, it sees whether any active transaction conflicts with T_i at site A. If not, T_i is *certified at site A*. If T_i is certified at all sites at which it accessed data, then it is 'really' certified; otherwise it is aborted.

A distributed SG certifier shares the problems of distributed SG schedulers. The certifier needs to check for cycles in a global graph every time a transaction ends.

2.5.5 Other Architectures

Centralized and distributed scheduling are endpoints of a spectrum. One can imagine hybrid architectures that feature multiple DMs per scheduler. See Figure 2.5. This architecture adds no technical issues beyond those already discussed.

Hierarchical scheduler architectures are also possible. See Figure 2.6. To our knowledge, no one has studied this approach.

2.6 Data Replication

In a *replicated database*, each *logical* data item, x, can have many *physical* copies denoted $\{x_1,\ldots,x_m\}$, which are resident at different DMs. Transactions issue Reads and Writes on logical data items. TM's translate those operations into Reads and Writes on physical data. The effect, as seen by each transaction, must be as if there were only one

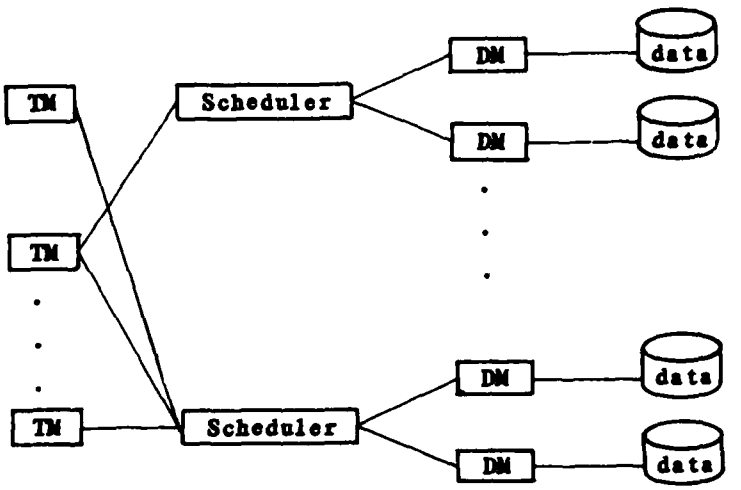


Figure 2.5 Hybrid Architecture

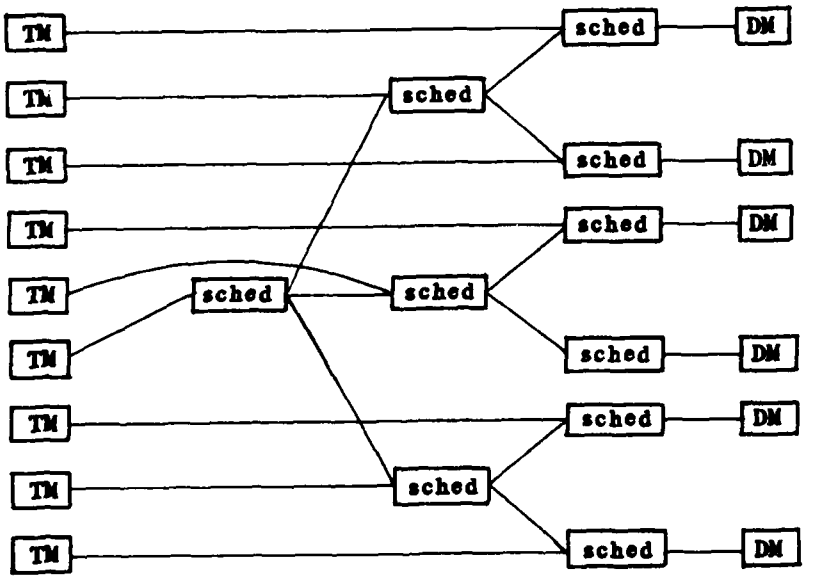
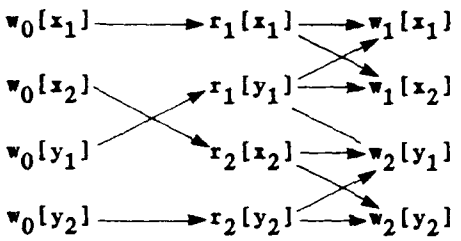


Figure 2.6 Hierarchical Architecture

copy of each data item.

There is a simple way to obtain this effect. Each TM translates $r_i[x]$ into $r_i[x_j]$ for some copy x_j of x and $w_i[x]$ into $\{w_i[x_j] | \text{all copies } x_j \text{ of } x\}$. If the scheduler(s) is SR, the effect is just like a nonreplicated database. To see this, consider a serial log equivalent to the SR log that executed. Since each transaction writes into <u>all</u> copies of each logical data item, each $r_i[x_j]$ read from the 'latest' transaction preceding it that wrote into any copy of x. But this is exactly what would have happened had there been only one copy of x. (For a more rigorous explanation, see [ATTA82].) Consider this example:

L_3 =



x_1 and x_2 are copies of logical data item x; y_1 and y_2 are copies of y. T_0 produces initial values for both copies of each data item. T_1 reads x and y, and writes x; T_2 reads x and y, and writes y.

L_3 is SR. It is equivalent to the following serial log:

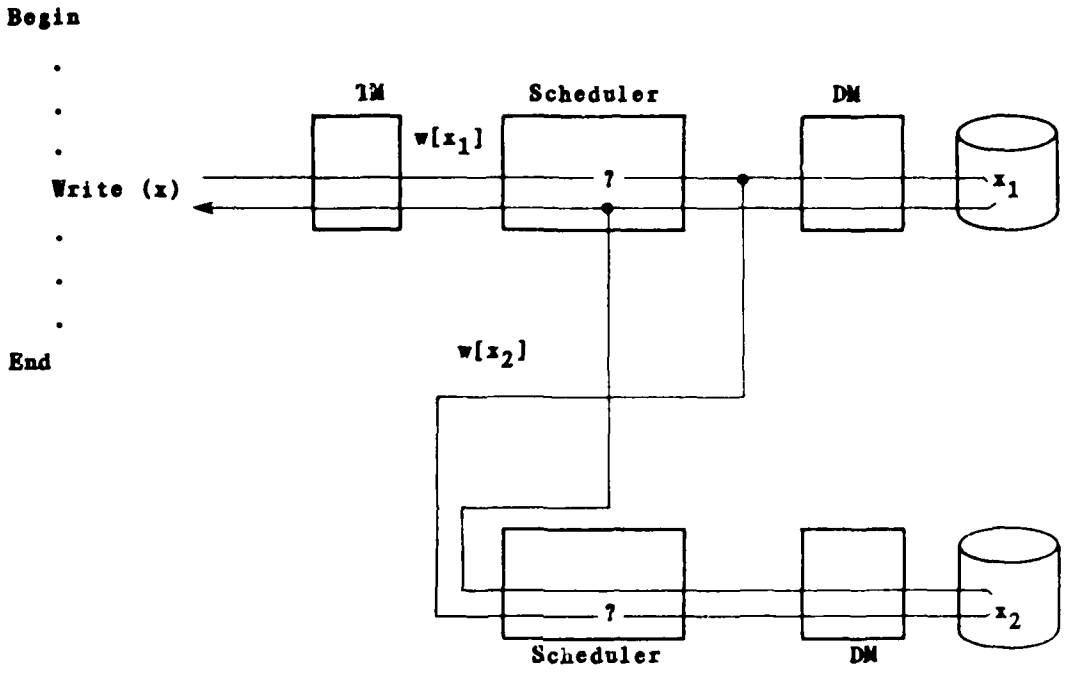
$$L_4 = w_0[x_1]\ w_0[x_2]\ w_0[y_1]\ w_0[y_2]\ r_1[x_1]\ r_1[y_1]\ w_1[x_1]\ w_1[x_2]\ r_2[x_2]\ r_2[y_2]\ w_2[y_1]\ w_2[y_2]\ .$$

Note that each Read, e.g., $r_2[x_2]$ or $r_2[y_2]$, reads from the 'latest' transaction preceding it that wrote into any copy of the data item. Therefore, L_4 has the same effect as the following log in which there is no replicated data:

$$L_4' = w_0[x]\ w_0[y]\ r_1[x]\ r_1[y]\ w_1[x]\ r_2[x]\ r_2[y]\ w_2[y]\ .$$

We call this the <u>do nothing</u> approach to replication -- just write into all copies of each data item and use an SR scheduler.

Two other approaches to replication have been suggested. In the <u>primary copy</u> approach, some copy of each x, say x_p, is designated as its primary copy [STON79]. Each TM translates $r_i[x]$ into $r_i[x_j]$ for some copy x_j, as before. Writes are translated differently. The TM



Note: x_1 is primary copy

Figure 2.7 Processing Writes in Primary Copy

translates $w_i[x]$ into a single Write, $w_i[x_p]$, on the primary copy. When the primary copy's scheduler outputs $w_i[x_p]$, it also issues Writes on the other copies of x (i.e., $w_i[x_1],\ldots,w_i[x_m]$). See Figure 2.7. These Writes are processed by the schedulers for $x_1,\ldots,x_m$ in the usual way. For example, in 2PL, the scheduler for x_j must put a write-lock on t_j for T_i before outputting $w_i[x_j]$. The primary copy's scheduler may be centralized (in which case the technique is called primary site [ALSB76]), or distributed with the primary copy's DM.

Primary copy is a good idea for 2PL schedulers. It eliminates the possibility of deadlock caused by Writes on different copies of one data item. Suppose x has copies x_1 and x_2. Suppose that T_1 and T_2 want to Write x at about the same time. In the do nothing approach, the following execution is possible: T_1 locks x_1; T_2 locks x_2; T_1 tries to lock

x_2 but is blocked by T_2's lock; T_2 tries to lock x_1 but is blocked by T_1's lock. This is a deadlock. Primary copy avoids this possibility because each transaction must lock the primary copy first.

In the voting approach to replication, TMs again distribute Writes to all copies of each data item [THOM79]. Assume that we are using distributed schedulers. When a scheduler is ready to output $w_i[x_j]$, it sends a vote of yes to the vote collector for x; it does not output $w_i[x_j]$ at this time. When the vote collector receives yes votes from a majority of schedulers, it tells all schedulers to output their Writes. (Each scheduler may need to update its local data structures before outputting $w_i[x_j]$ (e.g., set a write-lock on x_j.)) Assume each scheduler is correct (i.e., produces an acyclic SG). Then, since every pair of conflicting operations was voted yes by some correct scheduler (both operations got a majority of yes's), the SG must be acyclic and the result is correct.

The principal benefit of voting is fault tolerance; it works correctly as long as a majority of sites holding a copy of x are running. See [THOM79, GIFF79] for details.

2.7 Multiversion Data

Let us return to a database system model where each logical data item is stored at one DM.

In a multiversion database each Write $w_i[x]$, produces a new copy (or version) of x, denoted x^i. Thus, the value of x is a set of versions. For each Read, $r_i[x]$, the scheduler selects one of the versions of x to be read. Since writes don't overwrite each other, and since Reads can read any version, the scheduler has more flexibility in controlling the effective order of Reads and Writes.

Although the database has multiple versions, users expect their transactions to behave as if there were just one copy of each data item. Serial logs don't always behave this way. For example:

$$w_0[x^0]\ r_1[x^0]\ w_1[x^1y^1]\ r_2[x^0y^1]\ w_2[y^2]$$

is a serial log, but its behavior cannot be reproduced with only one copy of x. We must therefore restrict the set of allowable serial logs.

A serial log is 1-copy serial (or 1-serial) if each $r_i[x^j]$ reads from the last transaction preceding it that wrote into any version of x. The above log is not 1-serial, because r_2 reads x from w_0, but $w_0[x^0] < w_1[x^1] < r_2[x^0]$. A log is 1-serializable (1-SR) if it's equivalent to a 1-serial log. 1-serializability is our correctness criterion for multiversion database systems.

All multiversion concurrency control algorithms (that we know of) totally order the versions of each data item in some simple way. A version order, $<<$, for L is an order relation over versions such that, for each x, $<<$ totally orders the versions of x.

Given a version order $<<$, define the multiversion SG w.r.t. L and $<<$ (denoted MVSG(L,$<<$)) as SG(L) with the following edges added:* for each $r_j[x^j]$ and $w_k[x^k]$ in L, if $x^k << x^j$ then include $T_k \rightarrow T_j$, else include $T_j \rightarrow T_k$.

MULTIVERSION THEOREM [BERN81a]. A multiversion log is 1-SR iff there exists a version order $<<$ such that MVSG(L,$<<$) is acyclic.

[]

This theorem enables us to prove multiversion concurrency control algorithms to be correct. We must argue that for every log L produced by the algorithm, MVSG(L,$<<$) is acyclic for some $<<$.

The types of multiversion schedulers that have been proposed fall into two classes that approximately correspond to timestamping and locking.

---

*Note that the two operations conflict (and produce an edge in SG(L)) if they operate on the same version and one of them is a write.

2.7.1 Multiversion Timestamping

Multiversion concurrency control was first introduced by Reed in his multiversion timestamping method [REED78]. In Reed's algorithm, each transaction had a unique timestamp. Each Read and Write carries the timestamp of the transaction that issued it, and each version carries the timestamp of the transaction that wrote it. The version is defined by $x^i \ll x^j$ if $TS(i) < TS(j)$.

Operations are processed 'first come, first served'.** However, the version selection rules ensure that the overall effect is as if operations were processed in timestamp order. To process $r_i[x]$, the scheduler (or DM) returns the version of x with the largest timestamp $\leq$ TS(i). To process $w_i[x]$, version x^i is created, unless some $w_j[x]$ and $r_k[x]$ have already been processed with $TS(j) < TS(i) < TS(k)$. If this condition holds, the Write is rejected.

An analysis of MVSG(L,>>) for any L produced by this method shows that every edge $T_i \rightarrow T_j$ is in timestamp order ($TS(i) < TS(j)$). Thus MVSG(L,<<) is acyclic, and so L is 1-SR.

2.7.2 Multiversion Locking

In multiversion locking, the Writes on each data item, x, must be ordered. We define $x^i \ll x^j$ if $w_i[x^i] < w_j[x^j]$. Each version is in the certified or uncertified state. When a version is first written, it is uncertified. Each Read, $r_i[x]$, read either the last (wrt<<) certified version of z or any uncertified version of x. When a transaction finishes executing, the database system attempts to certify it. To certify T_i, three conditions must hold:

C1. For each $r_i[x^j]$, x^j is certified.

C2. For each $w_i[x^i]$, all $x^j \ll x^i$ are certified.

C3. For each $w_i[x^i]$ and each $x^j \ll x^i$, all transactions that read x^j have been certified.

These conditions must be tested atomically. When they hold, T_i is

---

**Handshaking is used to ensure that logically conflicting operations are executed by DMs in the order the scheduler output them.

declared to be certified and all versions it wrote are (atomically) certified.

An analysis of MVSG(L,<<) for any L produced by this method shows that every edge $T_i \rightarrow T_j$ is consistent with the order in which transactions were certified. Since certification is an atomic event, the certification order is a total order. Thus, MVSG(L,<<) is acyclic, and so L is 1-SR.

Two details of the algorithm require some discussion. First, the algorithm can deadlock. For example, in this log:

$$w_0[x^0]\ r_1[x^0]\ r_2[x^0]\ w_1[x^1]\ w_2[x^2]$$

T_1 and T_2 are deadlocked due to certification condition C3. As in 2PL, deadlocks can be detected by cycle detection on a waits-for graph whose edges include $T_i \rightarrow T_j$ such that T_i is waiting for T_j to become certified (so that T_i will satisfy C1-C3).

Second, C1-C3 can be tested atomically without using a critical section. Once C1 or C2 is satisfied for some $r_i[x^j]$ or $w_i[x^i]$, no future event can falsify it. When C3 becomes true for some $w_i[x^i]$, we 'lock' x^i so that no future reads can read versions that precede x^i. This allows C1-C3 to be checked one data item at a time. Of course, the waits-for graph must be extended to account for these new version locks.

Two similar multiversion locking algorithms have been proposed which allow at most one certified version of each data item. In Stearns' and Rosenkrantz's method [STEA81], the waits-for graph is avoided by using a timestamp-based deadlock avoidance scheme. In Bayer et al.'s method [BAYE80a, BAYE80b], a waits-for graph is used to prevent deadlocks. This algorithm consults the waits-for graph before selecting a version to read, and it always selects a version that creats no cycles.

Multiversion locking algorithms in which queries (read-only transactions) are given special treatment are described in [CHAN82, DUBO82, BERN82].

2.8 Combining the Techniques

The techniques described in Sections 2.4-2.8 can be combined in almost all possible ways. The three basic scheduling techniques (2PL, T/O, SG checking) can be used in scheduler mode or certifier mode. This gives six basic concurrency control techniques. Each technique can be used for rw or ww scheduling or both ($6^2 = 36$). Schedulers can be centralized or distributed ($36 \times 2 = 72$), and replicated data can be handled in three ways (Do Nothing, Primary Copy, Voting) ($72 \times 3 = 216$). Then, one can use multiversions or not ($216 \times 2 = 432$). By considering the multifarious variations of each technique, the number of distinct algorithms is in the thousands.

To illustrate our framework, we describe some of the algorithms that already have appeared in the literature.

The distributed locking algorithm proposed for System R* uses a 2PL scheduler for rw and ww synchronization. The schedulers are distributed at the DMs. Replication is handled by the do nothing approach.

Distributed INGRES uses a similar locking algorithm [STON79]. The main difference is that distributed INGRES uses primary copy for replication.

SDD-1 uses conservative T/O for rw scheduling and Thomas' write rule for ww scheduling. The algorithm has distributed schedulers and takes the do nothing approach to replication [BERN80b]. SDD-1 also uses conflict graph analysis, a technique for preanalyzing transactions to determine which run-time conflicts need not be synchronized.

A method using 2PL for rw scheduling and Thomas' write rule for ww scheduling is described in [BERN81b]. Distributed schedulers and the do nothing approach to replication were suggested. To ensure that the locking order is consistent with the timestamp order, one can use a Lamport clock: Each message is timestamped with the local clock time when it was sent; if a site receives a message with a timestamp, TS, greater than its local clock time, the site pushes its clock ahead to TS. After a transaction obtains all of its locks, it is assigned a timestamp using the TM's local Lamport clock. Thomas' majority consensus algorithm was

one of the first distributed concurrency control algorithms. It uses a 2PL certifier for rw scheduling and Thomas' write rule for ww scheduling. Schedulers are distributed and voting is used for replication. Each transaction is assigned a timestamp from a Lamport clock when it is certified. This ensures that the certification order (produced by rw scheduling) is consistent with the timestamp order used for ww scheduling.

Each of these algorithms is quite complex. A complete treatment of each would be lengthy. Yet, by understanding the basic techniques and how they can be correctly combined, we can explain the essentials of each algorithm in a few sentences.

Performance of these algorithms has been studied in [LIN81, LINN82a, LINN82b, LINN82c, LINN83, GARC78, GARC79, GELE78]. A comprehensive comparison of these algorithms can be found in Section 4.

3. Database Recovery Algorithms

3.1 Introduction

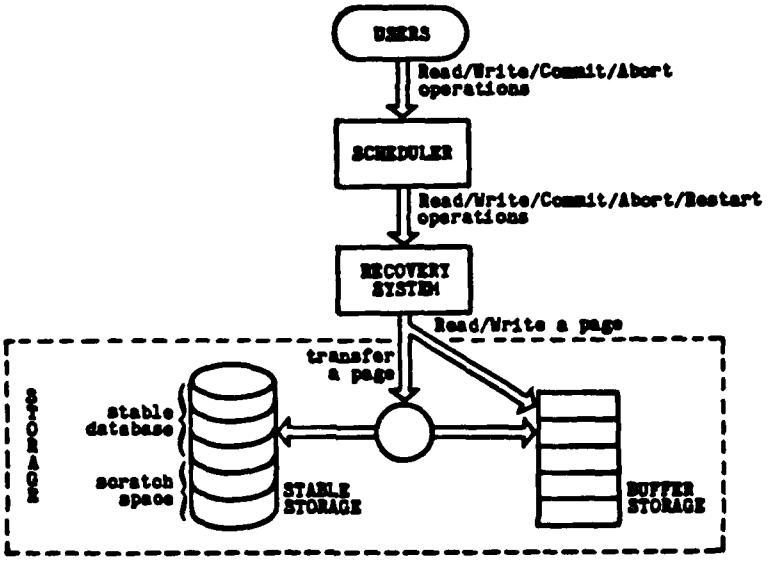
A database system (DBS) processes read and write commands issued by users' transactions to access the database. If a transaction fails in midstream, or if the system fails, the database may be left in an incorrect state. For example, if a money transfer transaction fails after posting its debit but before posting its corresponding credit, then the accounts are left unbalanced. The recovery algorithm of a DBS avoids these incorrect states by ensuring that the database only includes updates that are produced by transactions that execute to completion. This section is a survey of recovery algorithms for centralized and distributed DBSs.

Computer systems can fail in many ways, only some of which are handled by DBS recovery algorithms. We limit our attention to clean failures in which a transaction, the system, or, in the case of a distributed DBS, one site of the system, simply stops running. We do not consider traitorous failures in which components continue to run but perform incorrect actions (see [DOLE82, PEAS80]). We further limit attention to soft failures in which the contents of main memory are lost, but the contents of secondary memory (disk) remain intact. We do not consider methods for recovering from disk failures, although methods similar to those in this section apply (see [GRAY81, GRAY81, HARD79, HARD82, LIND79, LORI77, VERH78]).

We describe a model of centralized DBS recovery in Section 3.2. We present four cannonical types of centralized DBS recovery algorithms in Sections 3.3 through 3.6. We describe recovery algorithms for distributed DBSs in Section 3.7.

3.2 A Model Of Centralized Database System Recovery

We model a centralized database system as a scheduler, a recovery system, and storage.



Storage

The storage component consists of buffer storage and stable storage. Both are divided into physical pages of equal and fixed size. Buffer storage models main memory. Buffer storage is relatively fast, but of limited capacity, and it doesn't survive system crashes. Stable storage models disk memory and it is relatively slow, of (almost) unlimited capacity, and it does survive crashes.

The database consists of a set of logical pages. We assume that one physical copy (usually the most up-to-date copy) of each logical page is stored in a portion of stable storage called the stable database. Other portions of stable storage may be used by the recovery system as nonvolatile scratch space in ways that will be described later.

Transactions

A transaction is a program that can read from or write into the database. A transaction can issue four types of commands: Read, Write, Commit, and Abort. Read causes a page to be read from the database.

Write causes a new copy of a logical page to be written into the database. Commit tells the system that the transaction has terminated and that all of its updated pages should be permanently reflected in the database. Abort tells the system that the transaction has terminated abnormally and that the pages it wrote into should be returned to their previous state. (Commit and Abort may be issued by a process controlling the transaction, rather than by the transaction itself.) A transaction can have only one Commit or Abort processed.

A transaction is active if it has begun executing but has not yet had its Commit or Abort processed.

Notation: Each command is subscripted by the transaction that issued it. For example, $Read_i(P_j)$ is a Read issued by transaction T_i on page j.

The Scheduler

The scheduler controls the order in which Reads, Writes, Commits, and Aborts are passed to the recovery system. Although the scheduler allows commands from different transactions to be interleaved, it guarantees that the resulting execution is serializable. An execution is serializable if the effect is exactly the same as if the transactions had been executed serially, one after the next, with no concurrency at all. Many scheduling algorithms for attaining serializability are discussed in Section 2. Versions of all of them are compatible with the recovery algorithms described in this section.

The scheduler also guarantees that the execution is recoverable. An execution is recoverable if, for each transaction T_i, T_i is not committed until, for each page read by T_i, the transaction that last wrote that page is committed.

Recoverability is needed to avoid errors such as the following. Suppose T_i reads a page P_k last written by T_j (which is still active), T_i writes another page P_l, and commits. Now, suppose T_j fails and is aborted. Aborting T_j causes its write on P_k to be undone, thereby rendering T_i's input invalid. But, since T_i cannot be aborted after having been committed, T_i's updates to P_l must remain in the database

even though its input P_k is invalid.

For definitions, we assume that the scheduler uses page-level two-phase locking (2PL) [EAGE81]. Before outputting $Read_i(P_j)$ (resp. $Write_i(P_j)$), the scheduler sets a read lock (resp. write lock) on page P_j for transaction T_i. Two transactions cannot concurrently own conflicting locks on the same page, where read locks conflict with write locks and write locks conflict with read and write locks. If the scheduler receives an operation for which it can't set the corresponding lock, it delays the operation until the lock can be set.

When the scheduler receives a $Commit_i$ or an $Abort_i$, it forwards the operation directly to the recovery system. When the recovery system acknowledges that the operation has been processed, the scheduler then releases all the locks held by T_i.

Two-phase locking ensures serializability (see [BERN82, ESWA76] for proofs). The version of 2PL presented above also ensures recoverability by requiring that a transaction hold its write locks until its Commit or Abort is processed.

The Recovery System

The recovery system processes the Read, Write, Commit, and Abort commands it receives from the scheduler. It also handles system failures.

A system failure can interrupt the DBS at any moment. It causes all processing to stop and the contents of buffer storage to be lost. After the system recovers, transactions that were active at the time of the failure cannot continue executing because the contents of main memory are now useless. Thus, after the failure and before processing any other commands, the recovery system processes the restart command, whose effect is to abort all active transactions.

To handle failures properly, it is essential that the Commit command be implemented in a single instruction, normally a page write. If it were to require more than one instruction, a system failure could interrupt a partially completed Commit, making it ambiguous whether the transaction should be aborted during restart. Said differently, each

transaction must always be in one of three states: active, committed, or aborted, and each state change must be implemented by an atomic instruction execution.

Structuring Scratch Space

There are several types of information that a recovery algorithm stores in stable scratch space. It may store the identifiers of transactions that have committed, called the commit list. In this case, the single instruction that implements $Commit_i$ is usually a write that adds T_i to the commit list. The recovery algorithm also may store a list of identifiers of transactions that are active, called the active list, and those that have aborted, called the abort list.

Recovery algorithms often store copies of pages that were recently written on an audit trail (sometimes called a journal or log). For each write processed by the recovery algorithm, the audit trail may contain the identifier of the transaction that performed the write, a copy of the newly written page (called an after-image), and a copy of the physical page in the stable database that was overwritten by the write (called a before-image). Different algorithms vary considerably in the information they keep on the audit trail and in how they structure that information.

Undo and Redo

Recovery algorithms also differ in the time at which they write pages into the stable database. They may perform such writes before, concurrently with, or after the atomic instruction that commits the transaction that last wrote those pages.

Suppose that a page written by an active transaction is written into the stable database before the transaction commits. If the transaction aborts due to a system or transaction failure, the recovery algorithm must undo the write by restoring the previous copy (before-image) of the page.

Suppose that a page written by an active transaction is not written into the stable database before the transaction commits. If a system failure occurs after the transaction commits but before the page is

written into the stable database, the recovery algorithm must redo the write by moving the page to the stable database.

In every recovery algorithm, the after-images produced by a transaction must be written to stable storage (the database or scratch space) before the transaction commits. This is called the commit rule. If it is violated, a system failure shortly after a transaction T_i commits could leave the recovery algorithm with no stable copy of T_i's after-images, making it impossible to redo T_i.

Every recovery algorithm must also obey the log ahead rule: if an after-image is written to the stable database before the transaction that wrote it commits, then the before-image of that page must first be written to the audit trail. Otherwise, a system failure could occur after the after-image is in the stable database but before the before-image is in the audit trail, in which case the write could not be undone.

Categorization of Recovery Algorithms

Recovery algorithms can be categorized based on the timing of updates to the stable database. There are four types of recovery algorithms Some may require undo but not redo, redo but not undo, both undo and redo, and neither undo nor redo. These types of algorithms are described in Sections 3.3-3.6.

3.3 Algorithms That Undo But Don't Redo

For each type of recovery algorithm, we present a generic algorithm based on our database system model, and then we list example implementations. We describe this generic version by explaining how each command is processed. In all of the algorithms, the first command processed for T_i should add T_i to the active list.

For each operation, we mark by '{Ack}' the point at which the recovery system can acknowledge to the scheduler that the operation has been completed. Sometimes the operation has additional work to do after the acknowledgement is sent.

$Read_i(P_j)$. Copy P_j from the stable database into a buffer. {Ack}

$Write_i(P_j)$. Copy the before-image of P_j (from the stable database) to the audit trail. {Ack} Then* (after the disk acknowledges the write in the audit trail), write the new copy of P_j into the stable database.

$Commit_i$. Make sure all pages written by T_i are in the stable database. Then write T_i into the commit list. {Ack} Then delete it from the active list.

$Abort_i$. Write T_i into the abort list. Then undo all of T_i's writes by reading their before-images from the audit trail and writing them back into the stable database. {Ack} Then, delete T_i from the active list.

Restart. Process $Abort_i$ for each T_i on the active list. {Ack}

In this algorithm, all pages written by a transaction are written into the stable database before the transaction commits. Thus, redo is never needed, but an abort may require undo.

It is actually not necessary to write an after-image into the stable database <u>immediately</u> after the before-image is written into the audit trail. The after-image could be left in buffer storage for awhile, provided it is written to the stable database before the transaction commits as required by the commit rule.

This algorithm obeys the log ahead rule in processing $Write_i(P_j)$; the before-image of P_j is written to the audit trail before the after-image is written to the stable database.

The order in which writes are applied to stable storage is quite sensitive in this (and most other) recovery algorithms. In this algorithm, for example, in processing $commit_i$ it is incorrect to delete T_i from the active list before writing it into the commit list.

Remember that a system failure can occur during the processing of a $Restart_i$. So Restart must also take care tc reload the current active

---

* In every algorithm, we use 'then' to mean 'wait for the previous step to complete before proceeding to the next step'.

list into stable storage in order that it will be resilient to an system failure (followed by another Restart).

After $Commit_i$ or $Abort_i$ has been processed, the audit trail copies of pages written by T_i are no longer needed and can be returned to free space. The algorithm for garbage collecting these audit trail pages depends principally on the audit trail's data structure. We will not discuss garbage collection issues for any of the recovery methods described in this section.

The Prime Algorithm

This type of recovery algorithm is used in a database system product offered by Prime Computers [DUBO82], and in the DDM database system being developed at CCA [RIES82].

In Prime's algorithm, each page in the stable database has a pointer to its before-image in the audit trail. Each before-image in the audit trail points, in turn, to the next older before-image of the same page. Also, each physical page carries the transaction identifier of the transaction that wrote that particular copy. And, for each active transaction there is a convenient way to obtain a list of all pages it has written.

The page pointers are used for two purposes. First, to process an Abort, the pointer in each stable database page makes it easy to undo the aborted transaction's writes. Second, they help avoid concurrency control conflicts between queries and updates, as follows.

A query is a read-only transaction. Reads issued by queries are not locked in the scheduler but are passed directly to the recovery system (without being delayed). When the recovery algorithm receives the first read issued by a query T_i, say $Read_i(P_j)$, it reads the commit list and then selects the newest copy on the chained list of P_j copies whose transaction identifier is on the commit list. Subsequent reads by T_i are processed in the same way, using the copy of the commit list that was read when the first $Read_i$ was processed. By reading in this way, queries see a consistent copy of the database, yet they do not set read locks that might delay update transactions.

Another undo/no-redo algorithm is described in [RAPP75].

3.4 Algorithms That Redo But Don't Undo

In the generic algorithm, each command is processed as follows.

$Read_i(P_j)$. If T_i previously wrote P_j, then copy the after-image of P_j into a buffer. Otherwise, copy P_j from the stable database into a buffer. {Ack}

$Write_i(P_j)$. Write the new value of P_j into the audit trail. {Ack}

$Commit_i$. Write T_i into the commit list. Then for each page written by T_i, copy the after-image from the audit trail into the stable database. {Ack} Then delete T_i from the active list.

$Abort_i$. Write T_i into the abort list. {Ack} Then delete it from the active list.

Restart. For each T_i that is on the active list but not on the commit list, process $Abort_i$. {Ack} For each T_j on the active list and the commit list, process $Commit_j$.

In this algorithm, pages written by a transaction are not written into the stable database until after the transaction commits. Thus, undo is never needed, but a Restart may require redo.

This algorithm obeys the commit rule because the after-image of pages written by T_i are stored on the audit trail before T_i commits. It also obeys the log ahead rule, since no after-image of a transaction is written into the stable database before it commits.

Implementations of this algorithm are described in [LAMP76, MENA79]. This type of recovery algorithm is used in the INGRES Database System [STON79] and in SDD-1 [BERN80b].

3.5 Algorithms That Redo And Undo

In this algorithm, commands are processed as follows.

$Read_i(P_j)$. If T_i previously wrote P_j, then copy the after-image of P_j into a buffer. Otherwise, copy P_j from the stable database into a buffer. {Ack}

$Write_i(P_j)$. Copy the before-image and the after-image of P_j into the audit trail. {Ack} Then, sometime later, write the after-image into the stable database.

$Commit_i$. Write T_i into the commit list. Then, for each page written by T_i, write the after-image into the stable database (if it hasn't already been done). {Ack} Then, delete T_i from the active list.

$Abort_i$. Write T_i into the abort list. Then, for each page written by T_i, if its after-image has already been written into the stable database, write its before-image into the stable database. {Ack} Then delete T_i from the active list.

Restart. For each T_i on the active list and the commit list, process $Commit_i$. For each T_i on the active list but not on the commit list, process $Abort_i$. {Ack}

Note that Abort may require undo and Restart may require redo.

This algorithm obeys the commit rule, since the after-image of each page written by T_i is written into the audit trail before T_i commits. It also obeys the log ahead rule, since the before-image of each page written by T_i is written into the stable database.

One can improve the performance of this algorithm by using a variation proposed by Gray [GRAY81]. Gray's algorithm processes commands as follows.

$Read_i(P_j)$. If T_i previously wrote P_j, check to see if the after-image is in buffer storage. If not, copy P_j from the stable database to a buffer. {Ack}

$Write_i(P_j)$. Copy the before-image of P_j into buffer storage unless it is already there. Write the after-image of P_j into buffer storage;

Another undo/no-redo algorithm is described in [RAPP75].

3.4 Algorithms That Redo But Don't Undo

In the generic algorithm, each command is processed as follows.

$Read_i(P_j)$. If T_i previously wrote P_j, then copy the after-image of P_j into a buffer. Otherwise, copy P_j from the stable database into a buffer. {Ack}

$Write_i(P_j)$. Write the new value of P_j into the audit trail. {Ack}

$Commit_i$. Write T_i into the commit list. Then for each page written by T_i, copy the after-image from the audit trail into the stable database. {Ack} Then delete T_i from the active list.

$Abort_i$. Write T_i into the abort list. {Ack} Then delete it from the active list.

Restart. For each T_i that is on the active list but not on the commit list, process $Abort_i$. {Ack} For each T_j on the active list and the commit list, process $Commit_j$.

In this algorithm, pages written by a transaction are not written into the stable database until after the transaction commits. Thus, undo is never needed, but a Restart may require redo.

This algorithm obeys the commit rule because the after-image of pages written by T_i are stored on the audit trail before T_i commits. It also obeys the log ahead rule, since no after-image of a transaction is written into the stable database before it commits.

Implementations of this algorithm are described in [LAMP76, MENA79]. This type of recovery algorithm is used in the INGRES Database System [STON79] and in SDD-1 [BERN80b].

this step must not overwrite the before-image. {Ack} Sometime later, write the before-image into the audit trail, leaving a copy of the after-image in buffer storage. The after-image may be written into the stable database any time after the before-image is written into the audit trail. Once the after-image is written both to the audit trail and the stable database, it may be removed from buffer storage.

$Commit_i$. After all the after-images of pages written by T_i have been written into the audit trail, write T_i into the commit list. {Ack}

$Abort_i$ and Restart are the same as the generic algorithm.

This algorithm obeys the log ahead rule because the before-image of each page is written in the audit trail before the after-image is written in the stable database. The commit rule is also satisfied since T_i's after-images are written into the audit trail before T_i commits.

When all after-images written by T_i have been written into the stable database, T_i can be deleted from the active list. This tells Restart that T_i does not need to be redone.

The main benefit of this algorithm is that the decision to write pages into stable storage is usually left to the database system's buffer management algorithm. The recovery algorithm writes into stable storage only when the commit or log ahead rule requires it.

A detailed implementation of this algorithm that incorporates checkpoints, and in which transactions write records instead of entire pages, appears in [LIND79].

3.6 Algorithms That Don't Undo Or Redo

In the generic algorithm, each command is processed as follows.

$Read_i(P_j)$. If T_i previously wrote P_j, then copy the after-image of P_j into a buffer. Otherwise, copy P_j from the stable database into a buffer. {Ack}

$Write_i(P_j)$. Write the after-image of P_j into the audit trail. {Ack}

$Commit_i$. In a single instruction, write the after-images of all pages written by T_i into the stable database and delete T_i from the active list. {Ack}

$Abort_i$. Write T_i into the abort list. {Ack} Then delete it from the active list.

Restart. For each T_i on the active list, process $Abort_i$. {Ack}

Unfortunately, this description isn't very informative because it relies on a magical instruction that implements commit without even using a commit list. Notice that if the magical instruction is available, then undo isn't needed because a transaction's after-images are not written into the stable database before it commits, and redo isn't needed because a transaction's after-images are written into the stable database in the instruction that commits the transaction.

We will describe an implementation of the Commit instruction similar to one presented in [LORI77].

Lorie's Shadow Page Algorithm

Assume that the stable database is partitioned into files $\{F_1,\ldots,F_2\}$, each of which is a sequence of logical pages. Each file, F_j, has a page table, PT_j, whose entries point to the pages of F_j. That is, $PT_j[k3]$ contains the address of the k-th page of F_j; this page is denoted P_{jk}. Assume that each page table fits on one page in the stable database. The stable database also contains in a fixed address a master record, M, that points to the n page tables; M[j] contains the address of PT_j.

Abort and Restart are processed as in the generic algorithm. Read, Write, and Commit are processed as follows.

For each file, F_j, the first Read or Write that T_i issues on a page of F_j causes the recovery algorithm to make a copy of PT_j in buffer storage, denoted PT_{ji}. For each page P_{jk} that T_i writes, $PT_{ji}[k]$ will

point to the after-image of that page in the audit trail. (The other entries in PT_{ji} are irrelevant.)

$Read_i(P_{jk})$. If T_i previously wrote P_j, then copy the after-image of P_j from address $PT_{jk}[k3]$ into a buffer. Otherwise, use M to find PT_j and copy P_{jk} from address $PT_j[k3]$ in the stable database into a buffer. {Ack}

$Write_i(P_j)$. Write the new copy of P_{jk} into the audit trail. Then assign $PT_{ji}[k]$ the address of that audit trail page. {Ack}

$Commit_i$. Copy M into buffer storage. For each file F_j that T_i wrote into, use (the buffer copy of) M to find PT_j and copy it into an empty page of buffer storage. (There are now two page tables for F_j connected to T_i: the buffer copy of PT_j that was just read, and PT_{ji}.) For each page P_{jk} that was written by T_i, assign to the buffer copy of $PT_j[k3]$ the contents of $PT_{ji}[k]$. Then, write PT_j into a <u>new</u> location in scratch space; denote this new copy of PT_j by PT_j'. Then, for each F_j that T_i wrote into, assign to (the buffer copy of) M[j3] the address of PT_j'. Then write M back to its fixed address in stable storage. {Ack}

The commit algorithm prepares a scratch copy of the page table (PT_j'). This is accomplished by assigning to M[j] the address of PT_j' for each file F_j that T_i wrote. By writing M back to the stable database, the old copies of the page table (PT_j) are replaced by the new ones (PT_j').

The instruction that commits T_i is the one that writes the updated M back into the stable database. Before this write, any read will use the old copy of M to read the before-image of any page written by T_i. After this write, it will read the after-image of any such page.

The recovery algorithm can commit only one transaction at a time. That is, Commit is a critical section. If two transactions were (incorrectly) to commit concurrently, each transaction might read a copy of PT_j into buffer storage, change the pointers to pages it wrote, and write that copy of PT_j to the audit trail. Thus, two copies of PT_j' would exist. Whichever transaction updated M first would lose its updates to PT_j, since they would be overwritten by the second transac-

tion when it installed its copy of PT_j' by updating M.

A version of Lorie's algorithm is implemented in System R's recovery manager [GRAY81].

3.7 Recovery In A Distributed Database System

A distributed database system (DDBS) consists of a set of sites connected by a network. Each transaction can read or write data stored at any of the sites.

We model a DDBS by a set of processes called data modules (DMs) and transaction modules (TMs). A DM is a centralized database system as defined in Section 3.2. It processes Reads and Writes on pages stored at that DM. It also processes Commits and Aborts, which permanently install or undo the writes of a transaction at that DM.

A TM interfaces transactions and DMs. Each transaction, T_i, submits commands to one TM, say TM_a. To process $Read_i$ or $Write_i$, TM_a simply sends the command to the DM that stores the data being read or written. Let $Active_i$ be the DMs at which T_i was active. To process $Abort_i$, TM_a must ensure that every DM in $Active_i$ processes $Abort_i$. To process $Commit_i$, TM_a should try to ensure that every DM in $Active_i$ processes $Commit_i$.

Unfortunately, TMs and DMs may fail at unpredictable times. TMs must process commands so that such failures never cause it to produce incorrect results.

We assume that process (i.e., TM and DM) failures are 'clean'. If a process does not produce an expected response to a message within a timeout period, then the process has really failed. If one process believes another process is down, then all processes believe that the process is down. And, when a process recovers, it recognizes that it has just recovered from a failure and runs a special 'reintegration protocol'. Mechanisms that support these assumptions are beyond the scope of this section. (See [ATTA82, HAMM80, PARK82, WALT82].)

Each TM keeps an active list, commit list, and abort list in stable storage. And, for each T_i on the active list, it maintains $Active_i$ in stable storage. When it receives a Read or Write from T_i, it sends the command to the appropriate DM and adds that DM to $Active_i$. For the first such Read or Write, the TM also adds T_i to its active list. It processes Abort and Commit as follows.

$Abort_i$. Add T_i to the abort list. Then, send $Abort_i$ to each DM in $Active_i$. Wait for every DM to acknowledge that it processed $Abort_i$. {Ack} Delete T_i from the active list.

$Commit_i$. Add T_i to the commit list. Then, send $Commit_i$ to each DM in $Active_i$. Wait for every DM to acknowledge that it processed $Commit_i$. {Ack} Delete T_i from the active list.

If a TM fails and later restarts, then it processes a Restart in the usual way. For each T_i on both the commit list and the active list, process $Commit_i$. For all other T_i on the active list, process $Abort_i$.

If a TM, say TM_a, discovers that a DM, say DM_b, has failed, then it normally processes $Abort_i$ for each T_i that has DM_b in $Active_i$. But what if DM_b is in $Active_i$ and TM_a has already sent $Commit_i$ to other DMs in $Active_i$? In this case it can't abort T_i because T_i already may be committed at some DMs. Instead, it must wait for DM_b to recover. When it does, TM_a sends $Commit_i$ to DM_b, too.

3.8 Two-Phase Commit

Each TM must obey the commit rule. That is, it must not send $Commit_i$ to any DM until every DM in $Active_i$ has T_i's after-images on stable storage. Otherwise, a DM in $Active_i$ may:

1. Fail before receiving $Commit_i$
2. Upon recovering, discover from TM_a that T_i has committed
3. But be unable to process $Commit_i$ because it lost some of T_i's after-images due to the failure

To obey the commit rule and thereby prevent (3), TM_a can use the two-phase commit protocol for processing Commit commands [LAMP76]. Phase one begins when TM_a receives $Commit_i$. It then sends a command called End_i to each DM in $Active_i$. A DM processes End_i by first ensuring that T_i's after-images at that DM are on stable storage and then sending an acknowledgement to TM_a. When TM_a has received the acknowledgement from every DM in $Active_i$, phase one is done. {Ack} In phase two, TM_a sends $Commit_i$ to each DM.

Since TM_a does not send $Commit_i$ to any DM until every DM has acknowledged End_i, no DM in $Active_i$ will process $Commit_i$ until every DM has T_i's after-image on stable storage.

If a DM, say DM_b, fails before acknowledging End_i, then TM_a won't leave phase one. Since TM_a cannot be sure that DM_b will be able to process $Commit_i$ when it recovers, TM_a must either wait for DM_b to recover or abort T_i by sending $Abort_i$ to every DM in $Active_i$. In practice, TM_a simply waits a prespecified timeout period after distributing the End_i's; if it hasn't received an acknowledgement of some End_i by this time, it assumes the DM has failed and aborts T_i.

Until a DM processes End_i, it may unilaterally decide to abort T_i by sending an $Abort_i$ command to TM_a. Once a DM acknowledges End_i, it loses its right to unilaterally abort T_i, and may only abort T_i if directed to do so by TM_a.

3.9 Three-Phase Commit

The TM algorithm presented above has a serious disadvantage. Suppose TM_a sends End_i to DM_b, DM_b acknowledges End_i, and then TM_a fails. Since DM_b doesn't know whether T_i will commit or abort, it has to wait for TM_a to recover. In particular, it must hold T_i's locks until TM_a recovers. If TM_a is supervising many active transactions, large portions of the database may be locked and unavailable until TM_a recovers.

We can avoid this problem by providing each TM with one or more backup TMs. If a TM fails, the backups can take over its functions.

One such algorithm is three-phase commit [SKEE81a, SKEE82a, SKEE82b, SKEE81b]. Each backup for TM_a maintains a commit list, CL_a. To process $Commit_i$ TM_a behaves as follows.

1. TM_a sends End_i to each DM in $Active_i$. Then it waits for all DMs to acknowledge their End_i's.

2. TM_a sends a command called $Precommit_i$ to each backup TM. A TM processes $Precommit_i$ by adding T_i to its copy of CL_a, and then sending an acknowledgement to TM_a. TM_a waits for all backups to acknowledge $Precommit_i$.

3. TM_a sends $Commit_i$ to each DM in $Active_i$.

Essentially, this is the two-phase commit protocol with a new phase added (Step 2).

If a backup TM fails, TM_a can ignore the failure if the number of backups is still acceptably large; otherwise, it should acquire another backup TM to replace the failed one.

Suppose TM_a fails. When the backups discover the failure, they elect one of their member TMs, say TM_b, to replace TM_a. After TM_b is elected, every other backup TM sends its copy of CL_a to TM_b. TM_b takes the union of those copies and distributes the result to other backups. This becomes everyone's copy of CL_a. When this process is complete, TM_b tells all DMs that it has taken over TM_a's functions.

If a DM wants to know what happened to a particular transaction, T_i, that was supervised by TM_a, it asks TM_b. If T_i is in TM_b's CL_a, then TM_b tells the DM to commit T_i; otherwise, it tells the DM to abort T_i. Thus, a transaction that was supervised by TM_a is committed if and only if it reached the second phase of three-phase commit and at least one of its precommits reached a backup TM (that didn't fail).

The algorithm for electing a backup TM to replace TM_a is easy, as long as none of the backups fail or recover from failure during the election. Assume each TM has a unique identifier. To elect a replacement for TM_a, each backup exchanges its identifier with every other backup. The TM with the largest identifier wins the election and takes

over.

If backup TMs fail or recover from failure during the election, the above algorithm can misbehave. Each of two TMs can conclude that it won the election. Algorithms to prevent this behavior are discussed in [GARC82, SKEE81a].

It is possible that TM_a and all of its backups fail during a short time period — too short for replacement backups to be acquired. This is called a total failure of TM_a; no TM can ever take over its function. DMs must wait until TM_a and enough of its backups have recovered so that the correct status of TM_a's transactions can be determined. Algorithms for recovering after total failure are discussed in [SKEE81].

Many variations on three-phase commit protocols have been proposed and analyzed. See [ALSB78, ALBS76, COOP82, EAGE81, HAMM80, LAMP78, MENA78, TRAI82].

3.10 Replicated Data

If a DM fails, transactions that need the failed DM's data must wait for the DM to recover. To avoid this delay, the DBS can replicate data; that is, it can store parts of the database at more than one DM. If one copy is unavailable due to a DM failure, other copies can be used instead.

Many concurrency control algorithms are known for keeping multiple copies of each page mutually consistent. However, even if concurrency control is performed correctly, failures can cause transactions to malfunction.

For example, suppose P_1 has copies P_{1a} and P_{1b} at DM_a and DM_b (resp.), and P_2 has copies P_{2c} and P_{2d} at DM_c and DM_d. T_1 reads P_1 and writes P_2; T_2 reads P_2 and writes P_1. Replicated data is handled by the 'intuitive' algorithm: to read data, read any copy; to write data, write all available copies. The following execution obeys these rules, yet it is incorrect.

$Read_1(P_{1a})$ DM_d-fails $Write_1(P_{2c})$

$Read_2(P_{2d})$ DM_a-fails $Write_2(P_{1b})$

This execution is incorrect because T_1 reads (a copy of) P_1 before T_2 writes P_1, while T_2 reads (a copy of) P_2 before T_1 writes P_2. The first condition means that T_1 appears to precede T_2, while the second condition means that T_2 appears to precede T_1. These conditions cannot both hold in a serial execution, and so the given execution is incorrect.

Algorithms for correctly processing commands on replicated data in the presence of DM failures appear in [ALSB78, ALBS76, COOP82, EAGE81, GIFF79, HAMM80, MENA78, THOM79]. No consensus on the best approach to this problem has yet emerged.

4. Performance of Distributed Concurrency Control

Many factors effect the performance of a distributed concurrency algorithm:

1. IO delay,
2. communication delay,
3. ratio of read-only to write transactions,
4. database size, transaction size,
5. system multiprogramming level,
6. distribution and replication of the database,
7. overhead of deadlock detection,
8. and system load, defined as the product of transaction size and multiprogramming level divided by the database size.

Our simulation study of the performance of distributed concurrency control algorithms shows that four of these factors have more significant impact than the others: IO delay, communication delay, transaction size, and system load. Hence we divide our simulation results into groups and discuss them separately by classifying the system environment as either IO-bound or communication bound, and as either short transaction loaded or long transaction loaded. We consider a system to be IO bound if queuing for IO or CPU resources is a more significant problem than queuing for communication channel; and we consider a system to be communication bound if queuing for communication channel is a more significant problem than queuing for IO and CPU resources. We consider a system to be short transaction loaded if the average number of data items requested by the transactions (or transaction size) is less than 0.05% of the database. The system is long transaction loaded if the average is larger than 0.2% of the database. If the average is between 0.05% and 0.2% of the database, the classification of the system as short transaction loaded or long transaction loaded depends on the system load. Details of the classification can be found in Figure 4.1.

Thus we present four categories of system environments: short transaction loaded and IO bound (SIO), short transaction loaded and communication bound (SCM), long transaction loaded and IO bound (LIO), and long

System Load / Trans Size	< 10%	> 10%
< 0.05%	Short	Short
0.05%<0.2%	Short	Long
> 0.2%	Long	Long

Trans Size: Average number of data items requested by a transaction as a percentage of the database size.
System Load: Trans Size multiplied by the multiprogramming level.
Database size: Total number of data items in the database.

Figure 4.1 System Classification
(Short Loaded or Long Loaded)

transaction loaded and communication bound (LCM). For each of these four environments, we compare the performance of various concurrency control algorithms, taking into consideration the factors that are not used to classify the system environment — i.e. multiprogramming level, ratio of read-only to write transactions, distribution and replication of the database.

We first describe, in Section 4.1, the distributed DBMS model that we use to evaluate these algorithms. We then define and describe, in Section 4.2, the concurrency control algorithms that we evaluate. We compare these algorithms in Section 4.3.1 through 4.3.4 for each of the four environments. In Section 4.4 we summarize the results of Section 4. Details of the simulation results can be found in the Appendix. e

To use this section as a design guide, a system designer must first classify his system environment, using the following three parameters. First, he must decide whether his system environment is IO bound or communication bound. Second, he must estimate the average number of data items, as a percentage of the total number of data items in the database, requested by a transaction (transaction size). Third, he must estimate the average system load, which is the product of the transaction size and the multiprogramming level of the system (number of transactions running concurrently). Using these three parameters and Figure 4.1, the designer can find his system classification. For each classification, he can find the comparison of various distributed concurrency control algorithms in Section 4.3.1 through Section 4.3.4.

4.1 Performance Model

We assume that there are two kinds of transactions: read-only transactions and write transactions (update transactions). Write transactions always read what they write, and write what they read. This assumption may seem restrictive, but it is a good approximation of real applications. Our earlier simulation results [LIN81a] showed that the total number of requests and the ratio of read-only requests to write requests active at any moment in the system have much greater impact on the system performance than the ratio of read-only to write transactions. Moreover our analysis shows that a more general assumption of transactions would not favor any concurrency control algorithm; thus for performance comparison of the algorithms, this assumption would not distort the results. To use the results of this section to evaluate the performance of a system that has transactions reading more than writing, the ratio of read-only to write transactions in the system can be adjusted upward.

A read-only transaction consists of a sequence of read-only requests, and each request reads a data item. A write transaction consists of a sequence of write requests (update requests), followed by a two-phase commit. Requests from a transaction are processed sequentially; another request is initiated only after the previous one has been successfully processed.

As described in Section 2, a distributed DBMS consists of TMs, schedulers, and DMs. Each transaction is managed by a TM, which sequences its requests and sends them to the appropriate scheduler to be processed. If the scheduler site is different from the TM site, a communication delay is incurred.

If a request is read-only, the scheduler requests a read lock for the requested data item (assuming that a two phase locking algorithm is used). Depending on the particular concurrency control algorithm used, some lock managers may grant the lock without checking whether the request conflicts with another transaction. Other lock managers may check for the conflict. If a conflict is found, the read-only request waits and incurs a blocking delay. Depending on the concurrency control

algorithm used, the scheduler may initiate a deadlock detection when blocking occurs, thus incurring processing and possibly communication overhead. When the lock for the requested data item is obtained, the scheduler sends the read-only request to the appropriate DM, and the read-only request incurs a processing delay. A read-only transaction ends after all its requests have been successfully processed.

A write request is processed in a manner similar to a read request, except that successful processing of all write requests of a transaction is always followed by a two-phase commit, and a write transaction ends after the two-phase commit is successfully processed (two-phase commit is the only reliability algorithm that we use in our simulation of concurrency control algorithm).

If timestamp based algorithms are used, a timestamp is assigned to each transaction, and requests from the transaction inherit the transaction timestamp. Each data item also has read and write timestamps that record the timestamps of the transactions that last read from (or write into) the data item. For all the timestamp algorithms that we have evaluated, the scheduler always resides at the site of a DM, and a request is always sent to the scheduler at the site where the data is to be accessed. When a scheduler receives a request, it compares the timestamp of the request with the read and write timestamp(s) of the data item, and it may or may not delay the request, depending on the particular algorithm used. If the request is not blocked, it is sent to the DM at the scheduler site, and the request incurs a processing delay.

We simulate both IO bound and communication bound system environments. In the IO bound environment, we explicitly simulate queuing for local processing, which combines cpu and IO processing. We differentiate between local processing of simple messages, such as lock request, lock release, and deadlock detection, and local processing of data requests. The latter needs more processing time than the former. In the IO bound environment, we do not simulate queuing for communication channels. Communication delay is simply simulated by a delay drawn from a probabilistic distribution.

In the communication bound environment, we explicitly simulate queuing for communication channels, but not for local processing resources. In some cases, we differentiate between message and data transmission. The latter takes longer than the former. We simulate local delay (combining IO and cpu processing) by drawing a random number from a probabilistic distribution.

The performance parameters that we use to compare distributed concurrency control algorithms include read throughput, write throughput, average read response time, and average write response time. Read throughput is the number of read-only requests successfully completed per time unit; read-only requests processed and subsequently aborted are not included. The write throughput is similarly defined. Read response time is measured from the time a read-only request is initiated by a TM to the time when the next read-only request of the same transaction is initiated by the same TM. Thus, it may include communication delay, blocking delay, and processing delay. Average read response time averages over the response times of all successfully completed read-only requests. Average write response time is similarly computed.

In addition to blocking delay, communication delay, and processing delay, other factors also affect average response times and throughputs (e.g., transaction abortion, deadlock detection, and multiple versions of data). The concurrency control algorithms evaluated in this section can be differentiated by the way they trade off these factors. Some algorithms trade longer blocking delay for fewer transaction abortions, and others trade reversely. Some trade more communication delay for less blocking delay, and others trade reversely. We describe these algorithms in the next section. In Section 4.3, based on the total throughput, we compare and rank these algorithms. Detailed data of the performance parameters can be found in the Appendix.

4.2 Description of Algorithms

The algorithms that we will consider are listed below. Selection of these algorithms is based on our earlier heuristic evaluation reported in [BERN81a]. The selected algorithms were shown to perform better than the algorithms discarded. Names of some algorithms are linked by the conjunctive 'and' (e.g. Primary Site and Primary Site). The term before the conjunctive describes the method used for read requests, and the term after the conjunctive describes the method used for write requests. These algorithms are described briefly in this section and summarized in Figure 4.2. Details of these algorithms can be found in the references.

1. Primary Site and Primary Site Two Phase Locking (C-C)
2. Primary Copy and Primary Copy Two Phase Locking (P-P)
3. Basic and Basic Two Phase Locking (B-B)
4. Basic and Primary Copy Two Phase Locking (B-P)
5. Basic and Primary Site Two Phase Locking (B-C)
6. DDM Multiple Version and Optimistic Two Phase Locking (DDM)
7. Basic and Optimistic Two Phase Locking (Opm)
8. Majority Consensus Timestamp (Maj)
9. Wait-Die Two Phase Locking (Die)
10. Basic Timestamp (BaT)
11. Multiple Version Timestamp (MvT)
12. Dynamic Timestamp (Dyn)

The SDD-1 algorithm is not explicitly covered because the Dynamic Timestamp algorithm is an improved version of it ([LIN79, [LIN81]). Neither is the Conservative Timestamp algorithm covered, because this algorithm essentially executes transactions serially in timestamp order. Thus it can perform better than other algorithms only when the transaction size is very large and the system load is extremely heavy and concurrent execution of transactions becomes counterproductive.

The Primary Site and Primary Site method is essentially a centralized two-phase locking method. All requests for read locks and write locks are sent to and processed by a designated primary site, which may use backup sites to improve resiliency. This method trades fewer transaction abortions for more transaction blocking, and it checks for lock

conflict as early as possible. It detects deadlock as early as possible, and it avoids distributed deadlock detection; but it has a bottleneck at the primary site.

The Primary Copy and Primary Copy method is a generalized version of the Primary Site and Primary Site method. All requests for read locks and write locks are sent to and processed by a designated primary copy site. However, primary copy sites for different data items may be different, thus distributed deadlock may occur. This method also trades fewer transaction abortions for more transaction blocking, and it checks lock conflict as early as possible. It requires distributed deadlock detection, but it may delay deadlock detection to reduce communication overhead.

The Basic and Basic method sets read locks and reads data locally if a local copy is available; otherwise it locks and reads the closest copy. It sets write locks globally. For each update request, an update lock is requested from all copies, and the update request is granted only after locks from all copies are obtained. This method trades faster read-only transaction response time for slower write transaction response time. It also trades more transaction blocking for fewer transaction abortions. It checks for lock conflict and deadlock as early as possible, and at the expense of more communication overhead.

The Basic and Primary Copy method processes read requests as the previous method does, but it requests write locks only from a designated primary copy. This method checks for most lock conflict as soon as possible, but it may delay distributed deadlock detection to reduce communication overhead. This method also trades fewer transaction abortions for more transaction blocking.

The Basic and Primary Site method is similar to the last method except that update lock requests are sent to a central site instead of to several primary copy sites. Thus deadlock detection is more centralized than in the previous method, and overhead is more centralized at the primary site.

The DDM [CHAN82a, CHAN82b] method avoids conflict between read requests and update requests by keeping several versions of each data

item. For each update request, DDM locks locally (if a local copy exists, or locks the closest copy). The update lock is propagated to other copies at transaction end. Detection of most conflicts among update requests is delayed until transaction end. Thus blocking delay is minimized for most write transactions at the expense of more transaction abortions at transaction end.

The Basic and Optimistic method sets read and update locks locally, if a local copy exists; otherwise it locks the closest copy. The update lock is propagated to all copies when the transaction that holds the update lock ends. Thus, distributed lock conflict checking and deadlock detection is delayed until a transaction ends. This algorithm reduces transaction blocking delay at the expense of more transaction abortions.

The Majority Consensus algorithm is similar to the Basic Optimistic algorithm. Each transaction has two phases: a read phase and a commit phase. During the read phase, a transaction reads locally if a local copy exists; otherwise it reads the closest copy. Timestamps of data items read by a transactions are recorded. During the commit phase, both read-only and update transactions must be certified by comparing the timestamps of the data read by each transaction to the transaction timestamp. Because of the certification step, read-only transactions require more communication overhead in this algorithm than in the Basic Optimistic algorithm. The details of the algorithm can be found in [BERN81a,THOM79]. If the algorithm is modified to favor read-only transactions so that read-only transactions need no certification, then it requires no more communication overhead than the Basic Optimistic algorithm. This algorithm checks for lock conflicts as late as possible, and it trades less transaction blocking for more transaction abortions.

In the Wait-Die algorithm, a unique sequence number is attached to every transaction. A transaction always locks locally if a local copy is available; otherwise it locks the closest copy. The locks are propagated to other copies when the transaction commits. Whenever a transaction is blocked by another transaction, the algorithm compares the sequence numbers of the two transactions. If the blocked transaction has a lower priority sequence number, it waits, otherwise it aborts. This algorithm checks local lock conflict as soon as possible, but it

checks distributed conflict at transaction end. It has no transaction deadlock (at the expense of more transaction abortions).

In the Basic Timestamp method, a read and a write timestamp are attached to each data item of the database. Each transaction that reads or updates the data item updates its read or write timestamp. Conflict is detected by comparing the timestamp of the transaction that reads or writes a data item with the timestamps of the data item, and not by comparing the timestamps of two transactions as done by the Wait-Die algorithm. This algorithm is similar to the Wait-Die algorithm because it also avoids transaction deadlock. Unlike the Wait-Die algorithm, it has no blocking delay and possibly has more transaction abortions. This algorithm may have fewer transaction abortion than the Wait-Die algorithm when most transactions are read-only, because it allows two transactions (a read-only and a write) to access the same data item simultaneously.

The Multiple Version Timestamp algorithm is a generalization of the previous algorithm. It keeps several versions of each data item in order to reduce conflict between read-only transactions and update transactions. Thus, this method trades more overhead of maintaining multiple data versions for fewer transaction abortions.

The Dynamic Timestamp algorithm [LIN79, LIN81] is an improved version of SDD-1 algorithm; it is unique among all the algorithms that we will compare for the following reasons. It requires transaction timestamps but not data item timestamps. It does not avoid transaction blocking; thus it trades more transaction blocking for fewer transaction abortions. But it uses preanalysis of transactions to reduce unnecessary transaction blocking. This algorithm may require a lot of communication overhead when many null write messages are needed [BERN82, LIN79, LIN81], and its performance may depend on system load [LIN81]. Thus it may perform poorly in some system environments.

The principal characteristics of these algorithms are summarized in Figure 4.2.

	B-B	P-P	C-C	B-P	BaT	MvT	DDM	Opm	Maj	Die	Dyn
blocking/abortion	b	b	b	b	a	a	a	a	a	m	b
lock conflict check	s	s	s	s	s	s	x	x	l	x	s
deadlock detection	s	l	s	l			l	l	l		
Scheduler	2	2	2	2	t	t	2,c	2,c	c	2	t
Location of Scheduler	d	d	cn	d	d	d	d	d	d	d	d
Data Replication	n	p	p	p	p	p	p	p	v	p	n

b: transaction blocking is preferred.
a: transaction abortion is preferred.
m: both blocking and abortion are used.
s: conflict or deadlock is checked as soon as possible.
l: conflict or deadlock is checked as late as possible.
x: local conflict is checked as soon as possible, but distributed conflict is checked at transaction end.
' ': the item does not apply.
2: two-phase locking scheduler.
t: timestamp scheduler.
c: certifier scheduler.
2,c: mixed 2-phase locking and certifier scheduler.
cn: centralized.
d: distributed.
n: do nothing.
p: primary copy.
v: voting.

Figure 4.2 Summary of Concurrency Control Algorithms

4.3 Performance Evaluation

4.3.1 Short Transaction Loaded & IO Bound

In this section we compare the performance of distributed concurrency control algorithms in a system environment in which most transactions are relatively short and IO resource is the performance bottleneck. The comparison of these algorithms is summarized in Figure 4.3. The comparison is based on actual simulation results except for the Wait-Die, Majority Consensus Timestamp, and Dynamic Timestamp algorithms. The evaluation of the Wait-Die algorithm is based on its similarity to the Basic Timestamp algorithm; the evaluation of the Dynamic Timestamp algorithm is based on the results of [LIN81]; and the evaluation of the Majority Consensus Timestamp algorithm is based on its similarity with the Basic Optimistic algorithm.

Figure 4.3 shows that five algorithms perform better than others: the Basic Timestamp, Multiple Version Timestamp, DDM, Optimistic, and Wait-Die algorithms.

In the short transaction loaded and IO bound environment, we found that transaction abortion is a better strategy than transaction blocking (i.e. it is better to abort than to wait). The abortion strategy is used by the Basic Timestamp and Multiple Version Timestamp algorithms, and to a large degree by the Wait-Die algorithm. We also found that it is better to delay lock conflict detection than to detect lock conflict early. Both the DDM and the Basic Optimistic algorithms use the delay strategy.

Although the DDM algorithm uses locking for write transactions, and the Optimistic algorithm uses locking for both read and write transactions, blocking occurs only among local transactions that access data from the same site. Transactions running at different sites never block each other. Write locks are propagated to other sites at transaction end, then conflicts among transactions running at different sites are detected and always result in transaction abortions. Therefore performance of these two algorithms is closer to those of timestamp algorithms than to those of two-phase locking algorithms. However, notice that the DDM and Basic Optimistic algorithms always abort transactions at transaction end, while the timestamp algorithms may abort transactions at an earlier phase of their execution.

These five algorithms perform equally well in most cases. The timestamp algorithms perform better than the DDM and Basic Optimistic algorithms when the database is fully redundant (thus read-only transactions complete quickly), the R/W ratio is high (probability of conflict among data requests is small), and local delay is large (local blocking delay is large and abortion at transaction end is expensive). However when the database is less redundant, the DDM and Basic Optimistic algorithms perform slightly better than the timestamp algorithms. Both read-only and write transactions require some remote data accesses and take longer to complete, and this causes the probability of conflict among transactions to rise and the timestamp algorithms to abort more transactions.

The Basic Timestamp algorithm performs as well as the Multiple Version Timestamp algorithm, and the latter requires more overhead and storage space for keeping multiple versions of data [LINN83]. Therefore

the Basic Timestamp algorithm is preferable to the Multiple Version Timestamp algorithm, unless the multiple versions of data are required in any case for database recovery and resiliency. Similarly, the difference in performance between the DDM and Basic Optimistic algorithms is very small, and the former needs higher overhead and more storage space for keeping multiple versions of data. The Basic Optimistic algorithm is preferable, unless the versions of data are required in any case for database recovery and resiliency.

The Wait-Die algorithm performs slightly worse than the Basic Timestamp algorithm when most transactions are read-only. When a read-only transaction conflicts with a write transaction, the timestamp algorithms never abort the read-only transaction, and they abort the write transaction only when a nonserializable execution may occur. However when most transactions are write transactions, the Wait-Die algorithm is preferred because it performs as well as the Basic Timestamp method and it needs no data item timestamps, which require storage space and processing overhead.

The Dynamic Timestamp algorithm performs best when most transactions are read-only, communication is fast, database is almost fully redundant, and preanalysis can be done on most transactions. In this environment, the fast protocols, R1, R1a, R1ab, and R1b [LIN79], LIN82] apply to most transactions. Assuming system conditions remain the same except that the database is not redundant, the Dynamic Timestamp algorithm still performs relatively well, because more efficient protocols (R2, R2a, R2ab, and R2b) apply to most transactions. These protocols are not as efficient as the group of R1 protocols, but they are relatively fast compared with R3 protocol. In all other cases, either when the communication is slow or when most transactions update the database, the Dynamic Timestamp algorithm is not efficient.

The Majority Consensus algorithm performs reasonably well, but not as well as the Basic Optimistic algorithm. The Majority Consensus algorithm as proposed in [THOM79] requires extra communication overhead for read-only transactions. If the algorithm is modified to favor read-only transactions, so that read-only transactions need not be certified, then it would perform as well as the Basic Optimistic algorithm.

To summarize, in this environment transaction abortion is a better strategy than transaction blocking, and delayed lock conflict checking is a better strategy than early lock conflict checking.

			B-B	P-P	C-C	B-P	BaT	MvT	DDM	Opm	Maj	Die	Dyn
R/W	L/C	Red											
low	*	full	6	4	5	3	1	1	1	1	2	1	3
low	low	full	6	4	5	3	1	1	1	1	2	1	3
high	low	full	6	4	5	3	1	1	1	1	3	1	3
high	high	full	4	4	5	3	1	1	2	2	3	2	1
high	high	part	6	6	7	5	3	3	1	2	3	4	2
high	low	part	5	5	6	4	2	2	1	1	2	3	2
low	*	part	6	4	5	4	2	2	1	1	2	2	3

Rank 1 is best and Rank 6 is worst.
Rank numbers have no absolute meaning. They only show relative performance across a row, not across a column.
R/W: Ratio of Read-only transactions to Write transactions
L/C: Ratio of Local delay to Communication delay, excluding queuing delay
Red: Redundancy of the database
* : Does not matter

Figure 4.3 Performance Comparison: Short Transaction Loaded & IO Bound

4.3.2 Short Transactions & Communication Bound

In this section we compare the performance of distributed concurrency control algorithms in a system environment in which most transactions are relatively short and communication channel is the performance bottleneck. The comparison of the algorithms is summarized in Figure 4.4. The comparison is based on actual simulation results except for the Wait-Die, Majority Consensus, and the Dynamic Timestamp algorithms. The evaluation of the Wait-Die algorithm is based on its similarity to the Basic Timestamp algorithm; the evaluation of the Dynamic Timestamp algorithm is based on the results of [LIN81]; and the evaluation of the Majority Consensus algorithm is based on its similarity to the Basic Optimistic algorithm.

Figure 4.4 shows that seven algorithms perform better than the others: Basic-Primary Copy, Basic Timestamp, Multiple Version Timestamp, DDM, Basic Optimistic, Wait-Die, and Dynamic Timestamp.

We found that transaction abortion, similar to the SIO environment, is a better strategy than transaction blocking, and that delayed lock conflict detection is a better strategy than early detection. However, because of the communication channel bottleneck, performance of the algorithms that require extra communication messages degrade in some cases.

The Basic Timestamp and Multiple Version Timestamp algorithms perform best in all cases. However, when the database is fully redundant, the DDM and Basic Optimistic algorithms perform just as well. Read-only transactions never incur communication delays, and write transactions incur communication delays only during the commit phase. Therefore transactions finish fast, blocking delay is shorter, and abortion at transaction end is less expensive.

The Majority Consensus algorithm, as proposed in [THOM79], does not perform well because of the extra communication messages required for read-only transactions. If the algorithm is modified to favor read-only transactions, so that read-only transactions need not be certified, the algorithm would perform as well as the Basic Optimistic algorithm.

The Wait-Die algorithm performs just as well as the timestamp algorithms in most cases. However, when most transactions are read-only, the Wait-Die algorithm unnecessarily aborts more read-only transactions than the timestamp algorithms, thus performing worse than the timestamp algorithms.

The DDM algorithm performs as well as the timestamp algorithms when the database is fully redundant. However, when the database is less redundant and most transactions are read-only, its performance degrades as shown in Figure 4.4. When the database is not fully redundant, read-only transactions require one extra communication message, which causes a long delay in a communication bound environment.

The Basic-Primary Copy algorithm performs 10% to 20% worse than the best algorithms in all cases, because it incurs extra communication messages when obtaining locks from the primary copies, and it uses transaction blocking instead of transaction abortion. The Dynamic Timestamp algorithm performs best when most transaction are read-only and can be

preanalyzed. In this environment, the most efficient protocols can be used and communication overhead for null-write messages is minimized.

Since the Basic Timestamp algorithm performs as well as the Multiple Version Timestamp algorithm, the former is preferable unless the multiple versions of data are required in any case for database recovery and resiliency. Similar observations apply to the DDM and Basic Optimistic algorithms [LINN83].

Our conclusion is that in this environment abortion is better than blocking, and that delayed lock conflict checking is better than early lock conflict checking. However, some algorithms that use these two strategies may not perform well in some cases because they require extra communication messages.

			B-B	P-P	C-C	B-P	BaT	MvT	DDM	Opm	Maj	Die	Dyn
R/W	L/C	Red											
low	*	full	5	4	4	3	1	1	1	1	3	1	3
high	low	full	5	6	4	3	1	1	1	1	5	2	2
high	high	full	5	6	4	2	1	1	1	1	5	2	2
high	low	part	5	6	7	3	1	1	4	2	5	2	2
low	low	part	5	4	6	2	1	1	2	1	3	1	4
high	high	part	4	5	6	2	1	1	3	1	5	2	2
low	high	part	5	4	6	3	1	1	2	1	3	1	4

Rank 1 is best and Rank 6 is worst.
Rank numbers have no absolute meaning. They only show relative performance across a row, not a column.
R/W: Ratio of Read-only transactions to Write transactions
L/C: Ratio of Local delay to Communication delay, excluding queuing delay
Red: Redundancy of the database
* : Does not matter

Figure 4.4 Performance Comparison: Short Transaction Loaded & Communication Bound

4.3.3 Long Transaction Loaded & IO Bound

In this section we compare the performance of distributed concurrency control algorithms in a system environment in which most transactions are relatively long and IO resource is the bottleneck. The comparison is summarized in Figure 4.5. The comparison is based on actual simulation results except for the Wait-Die and Majority Consensus algorithms. The evaluation of the Wait-Die algorithm is based on its

similarity to the Basic Timestamp algorithm; and the evaluation of the Majority Consensus algorithm is based on its similarity to the Basic Optimistic algorithm.

Figure 4.5 shows that three algorithms perform better than the others: Basic Primary, DDM, and Basic-Optimistic.

In this environment (long transactions, heavy system load) transactions conflict with each other more often, but only a fraction of the conflicts lead to transaction deadlocks. Thus, transaction blocking is better than indiscriminate transaction abortion. Moreover, prompt lock conflict detection is better than procrastination. Lock conflicts that are detected at transaction end always lead to deadlocks. The Basic Primary, DDM, and Basic Optimistic algorithms use the blocking strategy. The Basic Primary algorithm uses the early lock conflict detection strategy.

The Basic Primary Copy algorithm performs best in this environment because it does not abort a transaction unless it deadlocks, and it detects lock conflicts as soon as they occur. However, when most transactions are read-only, and the database is not fully redundant, the Basic Primary Copy does not perform as well as the DDM and Basic-Optimistic algorithms, because the extra communication messages required by the Basic Primary Copy algorithm for write-locks and deadlock detections does not outweigh the extra transaction abortions by the DDM and Basic-Optimistic algorithm.

The DDM and the Basic Optimistic algorithms perform well in partially redundant databases, because more lock conflicts are detected during the reading phase of transactions and less transactions abort at the commit phase. However, when the database is fully redundant, most lock conflicts are detected during the commit phase, which always leads to deadlocks and transaction abortions, thus resulting in the poorer performance of these two algorithms in this conditions.

The timestamp algorithms do not perform as well as the Basic-Primary method because transaction blocking is better than transaction abortion. However, the timestamp algorithms perform better than the DDM and Basic-Optimistic algorithms, when the database is fully redundant.

Read-only transactions incur no communication delay and complete quickly; the read-phase of write transactions also completes quickly. Thus conflict between read-only transactions and write transactions that result in the abortion of write transactions is reduced. In addition, when the database is fully redundant, the timestamp algorithms detect more conflicts at the read-phase, thus aborting more transactions at earlier stages of processing, while the DDM and Basic-Optimistic algorithms detect more conflicts at the commit phase, thus aborting more transactions at their ends. However, when the database is not fully redundant, the DDM and Basic-Optimistic algorithms detect more conflicts at the read-phase, and they abort more transactions at the early stages of processing, thus performing better than the timestamp algorithms.

The Wait-Die algorithm performs as well as the Basic Timestamp algorithm, except when most transactions are read-only. Then the Basic Timestamp algorithm has higher throughput of read-only transactions than the Wait-Die algorithm.

The Majority Consensus algorithm also performs poorly because it delays lock conflict detection until transaction end, thus resulting in many late transaction abortions. In fact, all certifier algorithms that certify transactions at transaction end perform badly in the long transaction environment. The Primary Site $ Primary Site (C-C) and the Primary Copy $ Primary Copy (P-P) algorithms also perform relatively well when the database is fully redundant. These two algorithms abort fewer transactions than the Basic Timestamp, Multiple Version Timestamp, DDM, and Basic Optimistic algorithms, and the savings in transaction abortions more than make up for the extra communication messages required by the two algorithms. The Basic-Basic algorithm does not perform as well because it requires many more communication messages than other algorithms.

To summarize, in this environment transaction blocking is better than transaction abortion, and early lock conflict detection is better than late detection.

			B-B	P-P	C-C	B-P	BaT	MvT	DDM	Opm	Maj	Die
R/(R+W)	Loc/Com	Redundant										
low	low	full	5	2	2	1	2	2	3	3	4	2
high	low	full	5	2	2	1	2	2	3	3	4	3
low	high	full	5	2	2	1	2	2	3	3	4	2
high	high	full	5	2	2	1	2	2	3	2	4	3
low	low	part	5	2	2	1	3	3	1	1	4	2
high	low	part	5	3	3	2	3	3	1	1	4	3
low	high	part	5	2	2	1	3	3	1	1	4	2
high	high	part	5	3	3	2	3	3	1	1	4	3

Rank 1 is best and Rank 6 is worst.
Rank numbers have no absolute meaning. They only show relative performance across a row, not a column.
R/W: Ratio of Read-only ransactions to Write transactions
L/C: Ratio of Local delay to Communication delay, excluding queuing delay
Red: Redundancy of the database
* : Does not matter

Figure 4.5 Performance Comparison: Long Transaction Loaded & IO Bound

4.3.4 Long Transactions & Communication Bound

In this section, we compare the performance of distributed concurrency control algorithms in a system environment in which most transactions are long and communication channel is the bottleneck. The comparison of these algorithms is summarized in Figure 4.6. The comparison is based on actual simulation results except for the Wait-Die and Majority Consensus algorithms. The evaluation of the Wait-Die algorithm is based on its similarity to the Basic Timestamp algorithm; and the evaluation of the Majority Consensus algorithm is based on its similarity to the Basic Optimistic algorithm.

Figure 4.6 shows that six algorithms perform better than the others: Basic Timestamp, Multiple Version Timestamp, DDM, Basic Optimistic, Basic Primary, and Wait-Die.

In this system environment (long transactions, heavy system load, and long communication delay) transactions conflict with each other more often, but only a fraction of the conflicts lead to deadlocks; thus, transaction blocking is better than indiscriminate transaction abortion. Moreover, early lock conflict detection is better than procrastination. Lock conflicts detected at transaction end always lead to deadlocks.

Read-only transactions incur no communication delay and complete quickly; the read-phase of write transactions also completes quickly. Thus conflict between read-only transactions and write transactions that result in the abortion of write transactions is reduced. In addition, when the database is fully redundant, the timestamp algorithms detect more conflicts at the read-phase, thus aborting more transactions at earlier stages of processing, while the DDM and Basic-Optimistic algorithms detect more conflicts at the commit phase, thus aborting more transactions at their ends. However, when the database is not fully redundant, the DDM and Basic-Optimistic algorithms detect more conflicts at the read-phase, and they abort more transactions at the early stages of processing, thus performing better than the timestamp algorithms.

The Wait-Die algorithm performs as well as the Basic Timestamp algorithm, except when most transactions are read-only. Then the Basic Timestamp algorithm has higher throughput of read-only transactions than the Wait-Die algorithm.

The Majority Consensus algorithm also performs poorly because it delays lock conflict detection until transaction end, thus resulting in many late transaction abortions. In fact, all certifier algorithms that certify transactions at transaction end perform badly in the long transaction environment. The Primary Site $ Primary Site (C-C) and the Primary Copy $ Primary Copy (P-P) algorithms also perform relatively well when the database is fully redundant. These two algorithms abort fewer transactions than the Basic Timestamp, Multiple Version Timestamp, DDM, and Basic Optimistic algorithms, and the savings in transaction abortions more than make up for the extra communication messages required by the two algorithms. The Basic-Basic algorithm does not perform as well because it requires many more communication messages than other algorithms.

To summarize, in this environment transaction blocking is better than transaction abortion, and early lock conflict detection is better than late detection.

The Basic Primary, DDM, Basic Optimistic, and to certain degree the Wait-Die algorithms use the blocking strategy; and the Basic Primary and Wait-Die algorithms detect lock conflicts as early as possible. In addition, because of long communication delay, algorithms requiring extra communication messages may not perform well even if they use transaction blocking instead of transaction abortion. The DDM and the Basic Primary algorithms require extra communication messages in some cases.

The Basic Primary Copy algorithm performs the best when the database is not fully redundant because it requires no more communication messages than the other algorithms, and because it causes fewer unnecessary transaction abortions. Even when the database is not fully redundant, if most transactions are write transactions and local delay is high relative to the communication delay, the Basic Primary Copy algorithm still performs better than the Basic Timestamp, Multiple Version Timestamp, DDM, and Basic-Optimistic algorithms, because the latter abort write transactions frequently. However, when the database is fully redundant, the Basic Primary Copy algorithm requires more communication messages than the Basic Timestamp, Multiple Version Timestamp, DDM, and Basic Optimistic algorithms. Thus, except for the cases above, the extra communication messages required by the Basic Primary Copy algorithm make its performance worse than that of the Basic Timestamp, Multiple Version Timestamp, DDM, and Basic-Optimistic algorithm in this communication bound environment.

The timestamp based algorithms perform best when the database is fully redundant, then read-only transactions incur no communication delay and complete quickly. The read phase of write transactions also completes quickly. When read-only transactions and the read phase of write transactions complete quickly, conflicts between read-only and write transactions that result in abortion of the write transactions is reduced. Thus, unnecessary transaction abortion is reduced.

The DDM method avoids conflicts between read-only transactions and write transactions, but it pays with more abortions of write transactions at transaction end. Thus, when most transactions are read-only, it performs very well. The higher throughput of read-only transactions make up for the extra abortion of write transactions. Notice that DDM

requires a extra round of communication messages for read-only transactions when the database is not fully redundant. Then its performance degrades.

The Basic-Optimistic algorithm also performs well when most transactions are read-only; then read-only transactions and the read phase of write transactions complete quickly. Otherwise it performs poorly because the system is eventually saturated with many long write transactions that later abort.

The Wait-Die algorithm performs as well as the Basic Timestamp algorithm when most transactions are write transactions, but not as well when most transactions are read-only transactions. Since the Wait-Die algorithm needs no overhead for maintaining data item timestamps, it is preferable to the timestamp based algorithms if most transactions are write transactions.

The Basic $ Basic, Primary Copy $ Primary Copy, and Primary Site $ Primary Site algorithms perform poorly because they require more communication messages than other algorithms. Communication overhead is expensive in this communication bound environment.

To summarize, in this environment transaction blocking is better than transaction abortion, and early lock conflict detection is better than late detection. However, some algorithms that use these two strategies may not perform well in some cases because they require extra communication messages.

4.4 Conclusion

We found that five of the twelve algorithms perform best in various system environmentss: Basic Timestamp, Multiple Version Timestamp, DDM, Basic Optimistic, and Basic-Primary algorithms.

When most transactions are short, concurrency control algorithms that abort conflicting transactions (such as Basic Timestamp, Multiple Version Timestamp algorithms) perform better than algorithms that block conflicting transactions (such as the Basic Primary algorithm). In this

R/(R+W)	Loc/Com	Redundant	B-B	P-P	C-C	B-P	BaT	MvT	DDM	Opm	Maj	Die
low	low	full	6	5	5	6	1	1	5	4	6	1
high	low	full	6	5	5	4	1	1	3	3	6	2
low	high	full	6	5	5	1	2	2	4	3	6	2
high	high	full	6	5	5	4	2	2	1	3	6	3
low	low	part	6	5	5	1	3	3	2	3	6	3
high	low	part	6	5	5	1	2	2	2	1	6	3
low	high	part	6	5	5	1	2	2	4	3	6	2
high	high	part	6	5	5	2	3	3	1	2	6	3

Rank 1 is best and Rank 6 is worst.
Rank numbers have no absolute meaning. They only show relative performance across a row, not a column.
R/W: Ratio of Read-only transactions to Write transactions
L/C: Ratio of Local delay to Communication delay, excluding queuing delay
Red: Redundancy of the database
* : Does not matter

Figure 4.6 Performance Comparison: Long Transactions & Communication Bound

environment, transactions conflict rarely; and when they do conflict, the blocking transactions tend to be longer than the average transaction size and blocking delay long [LINN83]. If a two-phase locking algorithm must be used, algorithms that delay lock conflict checking (such as the DDM and the Basic Optimistic algorithms) perform better than those that expedite lock conflict checking (such as the Basic Primary algorithm). Unless the communication bandwidth is very high, communication delay can devastate system performance; thus, the designer should reduce communication delay by locally controlling and accessing data as much as possible.

The issue of balancing communication delay against data distribution and replication is part of the complex problem of distributed database design. Distributed database design must also take into account the issues of distributed query processing and distributed database reliability, and is beyond the scope of this handbook.

Behavior of systems that have long transactions is very different from that of systems that have short transactions. Long transactions degrade system performance very quickly because they have more transaction conflicts. Since only a fraction of these conflicts results in deadlocks, concurrency control algorithms that use transaction blocking

often perform better than those that use transaction abortion indiscriminately. Moreover, concurrency algorithms that detect transaction conflict earlier often perform better than those that detect transaction conflict later. The effect of communication delay on the performance of a system that has long transactions is even more devastating than the effect on a system that has short transactions. Thus the designer must reduce communication delay as much as possible by controlling and accessing data locally.

However, no matter which concurrency algorithm the designer uses, a system that has long transactions always performs worse than a system that has short transactions. The designer should design transactions to access as much data in parallel as possible, and to break long transactions into shorter transactions. Long transactions that cannot be broken into shorter ones must be executed in background mode.

Our performance study shows that no one algorithm performs best in all system and application environments. If the system environment is stable, the database designer can select one algorithm that performs best in the environment. If the system environment is not stable, the database designer can assign different weights to different environments according to how often the system stays in each environment. The database designer then selects the algorithm that has the best weighted average performance.

From the results, we can also conclude that the best algorithm would be one that could be adjusted by the system administrator according to the environment. The administrator would adjust the algorithm to use transaction abortion and delay lock conflict detection whenever transactions are short, and to use transaction blocking and detect lock conflicts as soon as possible whenever transactions are long. The adjustable algorithm would also alternate, depending on the load on the communication channel, between algorithms that have more localized control and algorithms that have more distributed control.

5. References

[AHO75] A.V. Aho, E. Hopcroft and J.D. Ullman, 'The Design and Analysis of Computer Algorithms,' Addison-Wesley Publishing Co. (1975).

[ALSB76] Alsberg, P.A. and J.D. Day, 'A Principle for Resilient Sharing of Distributed Resources,' Proc. 2nd Int. Conf. on Software Engineering, October 1976.

[ALSB78] Alsberg, P.A., G.G. Belford, J.D. Day and E. Grapa, 'Multi-copy Resiliency Techniques,' Distributed Data Management (J.B. Rothnie, P.A. Bernstein, D.W. Shipman, eds.), IEEE, 1978, pp. 128-176.

[ANDL82] Andler, S., I. Ding, K. Eswaran, C. Hauser, W. Kim, J. Mehl, R. Williams, 'System D: A Distributed System for Availability,' Proc. 8th VLDB, Sept. 1982, pp. 33-44.

[ATTA82] Attar R., P.A. Bernstein and N. Goodman, 'Site Initialization, Recovery, and Back-up in a Distributed Database System,' Proc. 6th Berkeley Workshop, Feb. 1982, pp. 185-202.

[BADA79] Badal, D.Z., 'Correctness of Concurrency Control and Implications in Distributed Databases,' Proc. COMPSAC 79 Conf., Chicago, Nov. 1979.

[BART82] Bartlett, J.F., 'A 'NonStop' Operating System,' in The Theory and Practice of Reliable System Design, (Siewiarek and Swarz, eds.), Digital Press, 1982, pp. 453-460.

[BAYE80a] Bayer, R., H. Heller and A. Reiser, 'Parallelism and Recovery in Database Systems,' ACM Trans. on Database Systems, Vol. 5, No. 2 (June 1980), pp. 139-156.

[BAYE80b] Bayer, R., E. Elhardt, H. Heller and A. Reiser, 'Distributed Concurrency Control in Database Systems,' Proc. Sixth Int. Conf. on Very Large Data Bases, IEEE, N.Y., 1980, pp. 275-284.

[BJOR72] Bjork L.A. and C.T. Davies, 'The semantics of the preservation and recovery of integrity in a data system,' IBM TR-02.540, Dec. 22, 1972.

[BERN78] Bernstein, P.A. J.B. Rothnie, N. Goodman and C.H. Papadimitriou, 'The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case),' IEEE Trans. on

Software Engineering, Vol. SE-4, No. 3 (May 1978).

[BERN79] Bernstein, P.A., D. Shipman and W.S. Wong, 'Formal Aspects of Serializability in Database Concurrency Control,' IEEE Trans. on Software Engineering, Vol. SE-5, No. 3, May 1979.

[BERN80a] Bernstein, P.A. and D. Shipman, 'The Correctness of Concurrency Mechanisms in a System for Distributed Databases (SDD-1),' ACM Trans. on Database Systems, Vol. 5, No. 1, March 1980.

[BERN80b] Bernstein, P.A., D.W. Shipman and J.B. Rothnie, 'Concurrency Control in a System for Distributed Databases (SDD-1),' ACM Trans. on Database Sys. 5, 1 (March 1980), pp. 18-51.

[BERN81a] Bernstein, P.A. and N. Goodman, 'Concurrency Control in Distributed Database Systems,' ACM Computing Surveys, 13, 2 (June 1981), pp. 185-221.

[BERN81b] Bernstein, P.A. N. Goodman and M.Y. Lai 'A Two-Part Proof Schema for Database Concurrency Control,' Proc. 1981 Berkeley Workshop on Distributed Databases and Computer Networks.

[BERN82] Bernstein, P.A. and N. Goodman, 'A Sophisticate's Introduction to Distributed Database Concurrency Control,' Proc. 8th VLDB, Sept. 1982, pp. 62-76.

[BERN83] Bernstein, P.A. and N. Goodman, 'Concurrency Control Algorithms of Multiversion Database Systems,' submitted for publication.

[BJOR73] Bjork, L.A, 'Recovery Scenario for a DB/DC System,' Proc. ACM Nat'l Conf., 1973, pp. 142-146.

[CASA79] Casanova, M.A. The Concurrency Control Problem of Database Systems, Lecture Notes in Computer Science, Vol. 116, Springer-Verlag, 1981 (originally published as TR-17-79, Center of Research in Computing Technology, Harvard University, 1979).

[CHAN82a] Chan, A., U. Dayal, S. Fox, N. Goodman, D. Ries and D. Skeen, 'Overview of an Ada Compatible Distributed Database Manager,' submitted for publication.

[CHAN82b] Chan, A. and R. Gray, 'Implementing Distributed Read-only Transactions,' submitted for publication.

[CHAN82] Chan, A., S. Fox, W.T. Lin, A. Nori, and D. Ries, 'The Implementation of an Integrated Concurrency Control and Recovery Scheme,' Proc. ACM SIGMOD Conf. on Management of Data, June 1982, pp. 184-191.

[CHEN80] Cheng, W.K. and G.C. Belford, 'Update Synchronization in Distributed Databases,' Proc. 2nd Berkeley Workshop on Very Large Data Bases, Oct. 1980.

[CHEN82] Cheng, W.K. and G.G. Belford, 'The Resiliency of Fully Replicated Distributed Databases,' Proc. 6th Berkeley Workshop, Feb. 1982, pp. 23-44.

[COOP82] Cooper, E.C, 'Analysis of Distributed Commit Protocols,' Proc. ACM SIGMOD Conf. on Management of Data, ACM, June 1982, pp. 175-183.

[DAVI73] Davies, C.T, 'Recovery Semantics for a DB/DC system,' Proc. ACM Nat'l Conf., 1973, pp. 136-141.

[DOLE82] Dolev, D, 'The Byzantine Generals Strike Again,' J. of Algorithms, 3, 1 (1982).

[DUBO82] Dubourdieu, D.J., 'Implementation of Distributed Transactions,' Proc. Sixth Berkeley Workshop on Distributed Data Management and Computer Networks, 1982, pp. 81-94.

[EAGE81] Eager, D.L, 'Robust Concurrency Control in a Distributed Database,' Univ. of Toronto TR CSRG #135, Oct. 1981.

[ESWA76] Eswaran, K.P., J.N. Gray, R.A. Lorie and I.L. Traiger, 'The Notions of Consistency and Predicate Locks in a Database System,' Commun. ACM, Vol. 19, No. 11, Nov. 1976, pp. 624-633.

[ELLI77] Ellis, C.A., 'A Robust Algorithm for Updating Duplicate Databases,' Proc. 2nd Berkeley Workshop on Distributed Databases and Computer Networks, May 1977.

[FISC82] Fischer, M.J. and A. Michael, 'Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network,' Proc. 1st ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, ACM, Mar. 1982, pp. 70-75.

[GALL82] Galler, B.I., Ph.D Thesis, Univ. of Toronto, 1982.

[GARC78] Garcia-Molina, H., 'Performance Comparisons of Two Update Algorithms for Distributed Databases,' Proc. 3rd Berkeley Workshop on Distributed Databases and Computer Networks, August 1978.

[GARC79a] Garcia-Molina, H., 'Performance of Update Algorithms for Replicated Data in a Distributed Database,' Ph.D. Dissertation, Computer Science Department, Stanford University, June 1979.

[GARC79b] Garcia-Molina, H., 'A Concurrency Control Mechanism for Distributed Data Bases Which Use Centralized Locking Controllers,' Proc. 4th Berkeley Workshop on Distributed Data Management & Computer Networks, August 1979.

[GARC82] Garcia-Molina, H, 'Elections in a Distributed Computing System,' IEEE Trans on Computers C-31, 1(Jan. 1982), pp. 48-59.

[GELE78] Gelenbe, E. and K. Sevcik, 'Analysis of Update Synchronization for Multiple Copy Data Bases,' Proc.3rd Berkeley Workshop on Distributed Databases and Computer Networks, August 1978.

[GIFF79] Gifford, D.K, 'Weighted Voting for Replicated Data,' Proc. 7th Symp. on Operating Systems Principles, ACM, Dec. 1979, pp. 150-159.

[GLIG80] Gligor, V.D. and S.H. Shattuck, 'On Deadlock Detection in Distributed Systems,' IEEE Trans. on Software Engineering, Vol. SE-6, No. 5, September 1980, pp. 435-440.

[GRAY75] Gray, J.N., R.A. Lorie, G.R. Putzulo and I.L. Traiger, 'Granularity of Locks and Degrees of Consistency in a Shared Data Base,' IBM Research Report RJ1654, September 1975.

[GRAY78] Gray, J.N., 'Notes on Database Operating Systems,' Operating Systems: An Advanced Course, Vol. 60, Lecture Notes in Computer Science, Springer-Verlag, N.,Y 1978, pp. 393-481.

[GRAY81] Gray, J.N., P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzulo and I. Traiger, 'The Recovery Manager of the System R Database Manager,' ACM Computing Surveys, 13, 2 (June 1981), pp. 223-242.

[HAMM80] Hammer, M.M. and D.W. Shipman, 'Reliability Mechanisms for SDD-1: A System for Distributed Databases,' ACM Trans. on Database Sys., Vol. 5, No. 5 (Dec. 1980), pp. 431-466.

[HARD79] Harder, T. and A. Reuter, 'Optimization of logging and recovery in a database system,' in Database Architecture, Bracchi and Nijssen, eds., North-Holland, 1979, pp. 151-168.

[HARD82] Harder, T. and A. Reuter, 'Principles of Transaction Oriented Database Recovery -- A Taxonomy,' Univ. Kaiserslautern TR 50/82.

[HOLT72] Holt, R.C., 'Some Deadlock Properties of Computer Systems,' Computing Surveys 4, 3 (Dec. 1972), pp. 179-195.

[KANE79] Kaneko, A., Y. Nishihara, K. Tsuruoka, and M. Hattori, 'Logical Clock Synchronization Method for Duplicated Database Control,' Proc. First Int'l. Conf. on Distributed Computing Systems, IEEE, N.Y., October 1979, pp. 601-611.

[KAWA79] Kawazu, S., S. Minamib, K. Itoh, and K. Teranaka, 'Two-Phase Deadlock Detection Algorithm in Distributed Databases,' Proc. 1979 Int'l. Conf. on Very Large Data Bases, IEEE, N.Y.

[KING74] King, P.F and A.J. Collmeyer, 'Database Sharing — An Efficient Method for Supporting Concurrent Processes,' Proc. 1974 NCC, AFIPS Press, Montvale, NJ, 1974.

[KIM79] Kim, K.H, 'Error Detection, Reconfiguration and Recovery in Distributed Processing Systems,' Conf. on Dist'd Computing, IEEE, 1979, pp. 284-294.

[KUNG79] Kung, H.T. and J.T. Robinson, 'On Optimistic Methods for Concurrency Control,' Proc.1979 Conf. on Very Large Data Bases, Rio de Janeiro, Brazil, October 1979.

[LAMP76] Lampson, B.W. and H. Sturgis, 'Crash Recovery in a Distributed Storage System,' Technical Report, Xerox PARC (1976)

[LAMP78a] Lamport, L, 'The Implementation of Reliable Distributed Multiprocess Systems,' Computer Networks, I 2 (1978), pp. 95-114.

[LAMP78b] Lamport, L., 'Time, Clocks and the Ordering of Events in a Distributed System,' Comm. of the ACM 21, 7, (July 1978), pp. 558-565.

[LAMP82] Lamport, L., R. Shostak and M. Pease, 'The Byzantine Generals Problem,' ACM Trans. on Programming Languages and Systems, Vol. 4, No. 3 (July 1982), pp. 382-401.

[LELA78] LeLann, G, 'Algorithms for Distributed Data-Sharing Systems Which Use Tickets,' Proc. 3rd Berkeley Workshop on Distributed Databases, and Computer Networks, August 1978.

[LIND79] Lindsay, B.G. et al., 'Notes on Distributed Databases,' IBM Research Report, No. RJ2571, July 1979.

[LIN79] Lin, W.K., 'Concurrency Control in a Multiple Copy Distributed Data Base System,' Proc. 4th Berkeley Workshop on Distributed Data Management & Computer Networks, August 1979.

[LIN81] Lin, W.K., 'Performance Evaluation of Two Concurrency Control Mechanisms in a Distributed Database System,' ACM SIGMOD-81 International Conference on Management of Data, April 1981, Ann Arbor, MI.

[LIN81a] Lin, W.K., et al, 'Distributed Database Control and Allocation First Semiannual Technical Report,' July 8, 1981, Computer Corp. of America, Cambridge, MA.

[LIN82a] Lin, W.K., et al, 'Distributed Database Control and Allocation Second Semiannual Technical Report,' Jan. 8, 1982, Computer Corp. of America, Cambridge, MA.

[LIN82b] Lin, W.K., et al, 'Distributed Database Control and Allocation Third Semiannual Technical Report,' July 8, 1982, Computer Corp. of America, Cambridge, MA.

[LIN83] Lin, W.K., et al, 'Distributed Database Control and Allocation Final Technical Report,' Feb. 8, 1983, Computer Corp. of America, Cambridge, MA.

[LINN82a] Lin, W.K. and J. Nolte, 'Performance of Two Phase Locking,' Proc. 1982 Berkeley Workshop on Distributed Data Management & Computer Networks, pp. 131-160.

[LINN82b] Lin, W.K. and J. Nolte, 'Read-Only Transaction and Two Phase Locking,' 2nd IEEE Symposium on Reliability in Distributed Software and Database Systems, July 1982, Pittsburgh, PA.

[LINN82c] Lin, W.K. and J. Nolte, 'Communication Delay and Two Phase Locking,' 3rd International Conference on Distributed Computing Systems, Oct. 1982, Fort Lauderdale, FL.

[LINN83] Lin, W.K. and J. Nolte, 'Basic Timestamp, Multiple Version Timestamp, and Two Phase Locking,' submitted for publication.

[LORI77] Lorie, R.A, 'Physical Integrity in a Large Segmented Database,' ACM Trans. on Database Sys., Vol. 2, No. 1 (Mar. 1977), pp. 91-104.

[MINO79] Minoura, T., 'A New Concurrency Control Algorithm for Distributed Data Base Systems,' Proc. 4th Berkeley Conf. on Distributed Data Management & Computer Networks, August 1979.

[MENA79] Menasce, D.A. and R.R. Muntz, 'Locking and Deadlock Detection in Distributed Databases,' IEEE Transactions on Software Engineering, Vol. SE-5, No. 3, May 1979, pp. 195-202.

[MENA80a] Menasce, D.A., G.J. Popek and R.R. Muntz, 'A Locking Protocol for Resource Coordination in Distributed Databases,' ACM Trans. on Database Sys., Vol. 5, No. 2, (June 1980), pp. 103-138.

[MENA80b] Menasce, D.A. and O.E. Landes, 'On the Design of a Reliable Storage Component for Distributed Database Management Systems,' Proc. 6th VLDB, Oct. 1980, pp. 365-375.

[MONT78] Montgomery, W.A., 'Robust Concurrency Control for a Distributed Information System,' Ph.D. dissertation, Laboratory for Computer Science, MIT, December 1978.

[PAPA77] Papadimitriou, C.A., Bernstein, P.A. and Rothnie, J.B., Jr., 'Some Computational Problems Related to Database Concurrency Control,' Proc. Conf. on Theoretical Computer Science, Waterloo, Ontario, August 1977.

[PAPA79] Papadimitriou, C.A., 'Serializability of Concurrent Updates,' Journal of ACM, Vol. 26, No. 4, Oct. 1979, pp. 631-653.

[PARK82] Parker, D.S. and R.A. Ramas, 'A Distributed File System Architecture Supporting High Availability,' Proc. 8th VLDB, Sept. 1982, pp. 161-184.

[PEAS80] Pease, M., R. Shostak and L. Lamport, 'Reaching Agreement in the Presence of Faults,' JACM, 27, 2 (1980), pp. 228-234.

[RAPP75] Rappaport, R.L, 'File Structure Design to Facilitate On-Line Instantaneous Updating,' Proc. of the 1975 SIGMOD Conf., pp. 1-14.

[REED78] Reed, D.P. , 'Naming and Synchronization in Decentralized Computer Systems,' Ph.D. Dissertation, MIT Department of Electrical Engineering, Sep. 1978.

[REED79] Reed, D.P, 'Implementing Atomic Actions,' Proc. 7th ACM Symp. on Operating Systems Principles, ACM, Dec. 1979.

[REUT80] Reuter, A, 'A Fast Transaction-Oriented Logging Scheme for Undo Recovery,' IEEE Trans on Soft. Eng., SE-6 (July 1980), pp. 348-356.

[RIES79a] Ries, D., 'The Effects of Concurrency Control on the Performance of a Distributed Data Management System,' Proc. 4th Berkeley Conf. on Distributed Data Management & Computer Networks, August 1979, Berkeley, CA, pp. 75-112.

[RIES79b] Ries, D., 'The Effect of Concurrency Control on Database Management System Performance,' Ph.D Thesis, Electronics Research Lab., Univ. Of CA, Berkeley, 1979.

[RIES82] Ries, D., A. Chan, U. Dayal, S.A. Fox, W.K. Lin and L. Yedwab, 'Decompilation and Optimization of ADAPLEX: A Procedural Database Language' Tech. Rep. CCA-82-04, Computer Corp. of America, Cambridge, MA. (in preparation 1982).

[ROSE78] Rosenkrantz, D.J., R.E. Stearns and P.M. Lewis, 'System Level Concurrency Control for Distributed Database Systems,' ACM Trans. on Database Systems, Vol. 3, No. 2, June 1978, pp. 178-198.

[SCHL79] Schlageter, G., 'Enhancement of Concurrency in DBS by the Use of Special Rollback Methods,' DB Architecture, Bracchi and Nijssen, eds., North-Holland, 1979, pp. 141-149.

[SHAP77] Shapiro, R.W. and R.E. Millstein, 'Reliability and Fault Recovery in Distributed Processing,' Oceans '77 Conf. Record, Vol. II, Los Angeles, CA, 1977.

[SILL80] Sillberschatz, A. and Z. Kedem, 'Consistency in Hierarchical Database Systems,' Journal of the ACM. Vol. 27, No. 1m, Jan 1980, pp. 72-80.

[SKEE81a] Skeen, D., 'Crash Recovery in a Distributed Database System,' Ph.D. Thesis, Dept. of Elec. Eng. and Comp. Sci., Univ. of CA, Berkeley, 1981.

[SKEE81b] Skeen, D. and M. Stonebraker, 'A Formal Model of Crash Recovery in a Distributed System,' Proc. 5th Berkeley Workshop, 1981, pp. 129-142.

[SKEE82a] Skeen, D, 'Nonblocking Commit Protocols,' Proc. 1982 ACM-SIGMOD Conf. on Management of Data, ACM, pp. 133-147.

[SKEE82b] Skeen, D, 'A Quorum Based Commit Protocol,' Proc. 6th Berkeley Workshop, Feb. 1982, pp. 69-80.

[STEA76] Stearns, R.E., P.M. Lewis, II and D.J. Rosenkrantz, 'Concurrency Controls for Database Systems,' Proc. of the 17th Annual Symposium on Foundations of Computer Science, IEEE, 1976, pp. 19-32.

[STEA81] Stearns, R.E. and D.J. Rosenkrantz, 'Distributed Database Concurrency Controls Using Before-Values,' Proc. 1981 ACM-SIGMOD Conf., ACM, N.Y., pp. 74-83.

[STRO81] Strom, B.I, 'Consistency of Redundant Databases in a Weakly Coupled Distributed Computer Conferencing System,' Proc. 5th Berkeley Workshop, 1981, pp. 143-153.

[STO79] Stonebraker, M., 'Concurrency Control and Consistency of Multiple Copies of data in Distributed INGRES,' IEEE Trans. on Soft. Eng., Vol. SE-5, No. 3, May 1979, pp. 188-194.

[THOM79] Thomas, R.H., 'A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases,' ACM Trans. on Database Systems, Vol. 4, No. 2, June 1979, pp. 180-209.

[TRAI82] Traiger, I.L., J. Gray, C.A. Galtier and B.G. Lindsay, 'Transactions and Consistency in Distributed Database Systems,' ACM Trans. on Database Systems, Vol. 7, No. 3, (Sept. 1982), pp. 323-342.

[VERH78] Verhofstad, J.M.S., 'Recovery Techniques for Database Systems,' ACM Computing Surveys, 10, 2 (1978), pp. 167-196.

[VERH79] Verhofstad, J.M.S, 'Recovery Based on Types,' DB Architecture, Bracchi and Nijssen, eds., North-Holland, 1979, pp. 125-139.

[WALT82] Walter, B, 'A Robust and Efficient Protocol for Checking the Availability of Remote Sites,' Proc. 6th Berkeley Workshop, Feb. 1982, pp. 45-68.

A.

Notations used in the appendix are explained here and in the figures.

READ THROUGHPUT: average number of read-only requests successfully processed per unit time (excluding requests processed and subsequently aborted).

WRITE THROUGHPUT: Average number of write requests successfully processed per unit time (excluding requests processed and subsequently aborted).

Average Response Per Read Request: average time required to process a read-only request.

Average Response Per Write Request: average time required to process a write request.

Basic Basic : Basic and Basic algorithm.
Prmry Prmry : Primary Copy and Primary Copy algorithm.
Cntrl : Primary Site and Primary Site algorithm.
Basic Prmry : Basic and Primary Copy algorithm.
Basic Cntrl : Basic and Primary Site algorithm.
Basic Tstmp : Basic Timestamp algorithm.
Mltpl Versn : DDM Multiple Version and Optimistic algorithm.
Basic Optms : Basic and Optimistic algorithm.

TZ=4, DZ=8192

MP	R/(R+W)	IO/Comm	no. of copy S1	S2	S3	Basic Basic	Prmry prmry	Cntrl	Basic Prmry	Basic Cntrl	Basic Tstmp	Mltpl Versn	Basic Optms
*	25%	.2	1	1	1	0.8	1.2	1.1	1.6		3.1	3.1	3.3
*	50%	.2	1	1	1	2.2	2.6	3.1					
*	75%	.2	1	1	1	6.6	4.5	8.3	15.		30	29	30
*	75%	1	1	1	1	6.3	4.4	5.9					
*	75%	.5	1	1	1	6.9	4.6	7.0	15.				
*	50%	.5	1	1	1				5.				
*	25%	.5	1	1	1				1.7				
*	75%	2	1	1	1				8.		9.8	9.8	9.2
*	50%	2	1	1	1				3.9				
*	25%	2	1	1	1				1.5		2.3	2.4	2.5
*	75%	.2	2/3	2/3	2/3	5.0	4.6	3.4	7.8		10.3	6.1	8.7
@	75%	.2	2/3	2/3	2/3			5.1					
#	75%	.2	2/3	2/3	2/3			5.9					
*	75%	.2	1/2	1/2	1/2	4.9	4.7	3.1					
*	50%	.2	2/3	2/3	2/3	2.4	2.8						
*	50%	.2	1/2	1/2	1/2	2.7	2.8						
*	25%	.2	2/3	2/3	2/3	0.9	1.3	.92	1.8		2.0	1.9	2.1
*	25%	.2	1/2	1/2	1/2	1.1	1.3	.88					
*	75%	.5	2/3	2/3	2/3			3.5	7.8				
*	75%	.5	1/2	1/2	1/2			3.0	6.4				
*	75%	1	2/3	2/3	2/3			3.2					
*	75%	1	1/2	1/2	1/2			3.0					
*	50%	.5	2/3	2/3	2/3				3.8				
*	50%	.5	1/2	1/2	1/2				3.6				
*	25%	.5	2/3	2/3	2/3				1.5				
*	25%	.5	1/2	1/2	1/2				1.5				
*	75%	2	2/3	2/3	2/3				6.2		6.9	5.4	6.6
*	75%	2	1/2	1/2	1/2				5.5				
*	50%	2	2/3	2/3	2/3				3.5				
*	50%	2	1/2	1/2	1/2				3.1				
*	25%	2	2/3	2/3	2/3				1.4		1.7	1.6	1.7
*	25%	2	1/2	1/2	1/2				1.4				
*	25%	.2	1	1/2	1/2			1.3					
*	50%	.2	1	1/2	1/2			3.5		5.0			
*	75%	.2	1	1/2	1/2			9.0		16.			
*	25%	.5	1	1/2	1/2			1.2					
*	50%	.5	1	1/2	1/2			3.3					
*	75%	.5	1	1/2	1/2			7.3					
*	25%	1	1	1/2	1/2			1.2					
*	50%	1	1	1/2	1/2			3.1		4.7			
*	75%	1	1	1/2	1/2			6.0		8.9			
*	25%	2	1	1/2	1/2			1.1					
*	50%	2	1	1/2	1/2			2.6					
*	75%	2	1	1/2	1/2			4.5					
@	50%	.2	1	1/2	1/2			4.8		5.4			
@	75%	.2	1	1/2	1/2			14.		17.			
@	50%	1	1	1/2	1/2			4.5		5.2			
@	75%	1	1	1/2	1/2			9.7		12.			
#	50%	.2	1	1/2	1/2			5.5		5.7			
#	75%	.2	1	1/2	1/2			16.		17.			
#	50%	1	1	1/2	1/2			5.4		5.7			
#	75%	1	1	1/2	1/2			13.		14.			

* Multiple programming levels at the three site are 10/11/11.
@ Multiple programming levels at the three site are 16/8/8.
# Multiple programming levels at the three site are 24/4/4.
TZ : Average no. of requests per transaction (transaction size).
DZ : Total number of data items in the database (database size).
MP : Multiprogramming level.
R/(R+W) : Percentage of transactions that are read-only.
IO/Comm : Ratio of local delay to communication delay (excluding queueing delay).
No. of Copy : Fraction of the database residing at sites S1, S2, & S3.

Figure A.1 READ THROUGHPUT: Short Transaction Loaded & Communication Bound

TZ=4, DZ=8192

MP	R/(R+W)	IO/Comm	no. of copy S1	S2	S3	Basic Basic	Prmry Prmry	Cntrl Total	Basic Prmry	Basic Cntrl	Basic Tstmp	Mltpl Versn	Basic Optms
*	25%	.2	1	1	1	2.2	3.5	3.4	5.1		9.6	9.4	9.4
*	50%	.2	1	1	1	2.2	2.6	3.0					
*	75%	.2	1	1	1	2.2	1.5	2.5	5		9.5	9.6	9.3
*	75%	1	1	1	1	2.1	1.5	1.9					
*	75%	.5	1	1	1	2.2	1.4	2.2	4.9				
*	50%	.5	1	1	1				4.9				
*	25%	.5	1	1	1				5.0				
*	75%	2	1	1					2.7		3.2	3.1	3.1
*	50%	2	1	1	1				4.0				
*	25%	2	1	1	1				4.5		7.4	7.2	7.3
*	75%	.2	2/3	2/3	2/3	1.7	1.6	1.3	2.7		3.4	2.0	3.1
@	75%	.2	2/3	2/3	2/3			1.7					
#	75%	.2	2/3	2/3	2/3			1.9					
*	75%	.2	1/2	1/2	1/2	1.7	1.6	1.0					
*	50%	.2	2/3	2/3	2/3	2.4	2.7						
*	50%	.2	1/2	1/2	1/2	2.6	2.8						
*	25%	.2	2/3	2/3	2/3	2.7	3.6	2.7	4.8		6.2	5.4	6.0
*	25%	.2	1/2	1/2	1/2	3.2	3.8	2.6					
*	75%	.5	2/3	2/3	2/3			1.1	2.6				
*	75%	.5	1/2	1/2	1/2			.98	2.2				
*	75%	1	2/3	2/3	2/3			1.2					
*	75%	1	1/2	1/2	1/2			.93					
*	50%	.5	2/3	2/3	2/3				4.0				
*	50%	.5	1/2	1/2	1/2				3.7				
*	25%	.5	2/3	2/3	2/3				4.7				
*	25%	.5	1/2	1/2	1/2				4.7				
*	75%	2	2/3	2/3	2/3				2.0		2.3	1.7	2.2
*	75%	2	1/2	1/2	1/2				1.8				
*	50%	2	2/3	2/3	2/3				3.3				
*	50%	2	1/2	1/2	1/2				3.2				
*	25%	2	2/3	2/3	2/3				4.3		5.2	4.8	5.2
*	25%	2	1/2	1/2	1/2				4.2				
*	25%	.2	1	1/2	1/2			3.9					
*	50%	.2	1	1/2	1/2			3.5		5.0			
*	75%	.2	1	1/2	1/2			2.9		5.2			
*	25%	.5	1	1/2	1/2			3.8					
*	50%	.5	1	1/2	1/2			3.2					
*	75%	.5	1	1/2	1/2			2.5					
*	25%	1	1	1/2	1/2			3.6					
*	50%	1	1	1/2	1/2			3.0		4.7			
*	75%	1	1	1/2	1/2			2.0		3.0			
*	25%	2	1	1/2	1/2			3.4					
*	50%	2	1	1/2	1/2			2.6					
*	75%	2	1	1/2	1/2			1.5					
@	50%	.2	1	1/2	1/2			4.8		5.4			
@	75%	.2	1	1/2	1/2			4.4		5.4			
@	50%	1	1	1/2	1/2			4.4		5.2			
@	75%	1	1	1/2	1/2			3.2		4.1			
#	50%	.2	1	1/2	1/2			5.5		5.7			
#	75%	.2	1	1/2	1/2			5.4		5.6			
#	50%	1	1	1/2	1/2			5.3		5.6			
#	75%	1	1	1/2	1/2			4.3		5.0			

* Multiple programming levels at the three site are 10/11/11.
@ Multiple programming levels at the three site are 16/8/8.
# Multiple programming levels at the three site are 24/4/4.
TZ : Average no. of requests per transaction (transaction size).
DZ : Total number of data items in the database (database size).
MP : Multiprogramming level.
R/(R+W) : Percentage of transactions that are read-only.
IO/Comm : Ratio of local delay to communication delay (excluding queueing delay).
No. of Copy : Fraction of the database residing at sites S1, S2, & S3.

Figure A.2 WRITE THROUGHPUT: Short Transaction Loaded & Communication Bound

TZ=4, DZ=8192

MP	R/(R+W)	IO / Comm	no. of copy S1	S2	S3	Basic Basic	Prmry Prmry	Cntrl Total	Basic Prmry	Basic Cntrl	Basic Tstmp	Mltpl Versn	Basic Optms
*	25%	.2	1	1	1	.31	5.1	4.0	.26		.20	.20	.22
*	50%	.2	1	1	1	.27	4.9	3.3					
*	75%	.2	1	1	1	.26	4.7	2.1	.26		.20	.20	.22
*	75%	1	1	1	1	1.1	4.6	3.3					
*	75%	.5	1	1	1	.55	4.7	2.7	.56				
*	50%	.5	1	1	1				.55				
*	25%	.5	1	1	1				.55				
*	75%	2	1	1	1				2.1		2	2	2
*	50%	2	1	1	1				2.1				
*	25%	2	1	1	1				2.1		2	2	2
*	75%	.2	2/3	2/3	2/3	2.7	4.6	6.2	2.5		2.2	2.6	2.5
@	75%	.2	2/3	2/3	2/3			4.9					
#	75%	.2	2/3	2/3	2/3			4.2					
*	75%	.2	1/2	1/2	1/2	3.6	4.6	7.2					
*	50%	.2	2/3	2/3	2/3	2.7	4.7	6.5					
*	50%	.2	1/2	1/2	1/2	3.7	4.6	7.4					
*	25%	.2	2/3	2/3	2/3	2.8	4.9		2.5		2.7	2.8	2.8
*	25%	.2	1/2	1/2	1/2	3.8	4.7						
*	75%	.5	2/3	2/3	2/3			6.3	2.6				
*	75%	.5	1/2	1/2	1/2			7.2	3.5				
*	75%	1	2/3	2/3	2/3			6.3					
*	75%	1	1/2	1/2	1/2			7.3					
*	50%	.5	2/3	2/3	2/3				2.6				
*	50%	.5	1/2	1/2	1/2				3.6				
*	25%	.5	2/3	2/3	2/3				2.7				
*	25%	.5	1/2	1/2	1/2				3.6				
*	75%	2	2/3	2/3	2/3				3.4		3.2	3.4	3.3
*	75%	2	1/2	1/2	1/2				4.1				
*	50%	2	2/3	2/3	2/3				3.5				
*	50%	2	1/2	1/2	1/2				4.2				
*	25%	2	2/3	2/3	2/3				3.6		3.4	3.5	3.5
*	25%	2	1/2	1/2	1/2				4.2				
*	25%	.2	1	1/2	1/2			3.5					
*	50%	.2	1	1/2	1/2			2.8		.63			
*	75%	.2	1	1/2	1/2			2.0		1.3			
*	25%	.5	1	1/2	1/2			4.0					
*	50%	.5	1	1/2	1/2			3.4					
*	75%	.5	1	1/2	1/2			2.5					
*	25%	1	1	1/2	1/2			4.4					
*	50%	1	1	1/2	1/2			3.7		1.9			
*	75%	1	1	1/2	1/2			3.2		2.7			
*	25%	2	1	1/2	1/2			4.9					
*	50%	2	1	1/2	1/2			4.6					
*	75%	2	1	1/2	1/2			4.3					
@	50%	.2	1	1/2	1/2			2.1		.40			
@	75%	.2	1	1/2	1/2			1.3		.59			
@	50%	1	1	1/2	1/2			2.7		1.4			
@	75%	1	1	1/2	1/2			2.2		1.8			
#	50%	.2	1	1/2	1/2			1.0		.32			
#	75%	.2	1	1/2	1/2			.71		.34			
#	50%	1	1	1/2	1/2			1.8		1.2			
#	75%	1	1	1/2	1/2			1.5		1.2			

* Multiple programming levels at the three site are 10/11/11.
@ Multiple programming levels at the three site are 16/8/8.
# Multiple programming levels at the three site are 24/4/4.
TZ : Average no. of requests per transaction (transaction size).
DZ : Total number of data items in the database (database size).
MP : Multiprogramming level.
R/(R+W) : Percentage of transactions that are read-only.
IO/Comm : Ratio of local delay to communication delay (excluding queueing delay).
No. of Copy : Fraction of the database residing at sites S1, S2, & S3.

Figure A.3 Average Response Per Read Request
Short Transactions & Communication Bound

TZ=4, DZ=8192

MP	R/(R+W)	IO / Comm	no. of copy S1	S2	S3	Basic Basic	Prmry Prmry	Cntrl Total	Basic Prmry	Basic Cntrl	Basic Tstmp	Mltpl Versn	Basic Optms
*	25%	.2	1	1	1	12	5.2	4.4	5.3		.2	.28	.29
*	50%	.2	1	1	1	12	5.1	3.5					
*	75%	.2	1	1	1	11	4.9	2.3	4.6		.2	.26	.26
*	75%	1	1	1	1	8.6	4.8	3.5					
*	75%	.5	1	1	1	10	4.8	2.7	3.9				
*	50%	.5	1	1	1				4.9				
*	25%	.5	1	1	1				5.2				
*	75%	2	1	1	1				4.5		2.0	2.2	2.1
*	50%	2	1	1	1				4.9				
*	25%	2	1	1	1				5.3		2.0	2.2	2.1
*	75%	.2	2/3	2/3	2/3	8.3	4.8	6.3	4.9		2.2	2.7	2.6
@	75%	.2	2/3	2/3	2/3			5.2					
#	75%	.2	2/3	2/3	2/3			4.5					
*	75%	.2	1/2	1/2	1/2	6.2	4.7	7.4					
*	50%	.2	2/3	2/3	2/3	8.3	4.9	6.9					
*	50%	.2	1/2	1/2	1/2	6.5	4.7	7.6					
*	25%	.2	2/3	2/3	2/3	8.4	5.0		5.0		2.7	2.9	2.9
*	25%	.2	1/2	1/2	1/2	6.5	4.8						
*	75%	.5	2/3	2/3	2/3			6.3	4.7				
*	75%	.5	1/2	1/2	1/2			7.5	4.8				
*	75%	1	2/3	2/3	2/3			6.5					
*	75%	1	1/2	1/2	1/2			7.5					
*	50%	.5	2/3	2/3	2/3				4.9				
*	50%	.5	1/2	1/2	1/2				4.8				
*	25%	.5	2/3	2/3	2/3				5.0				
*	25%	.5	1/2	1/2	1/2				4.9				
*	75%	2	2/3	2/3	2/3				4.9		3.2	3.6	3.5
*	75%	2	1/2	1/2	1/2				4.9				
*	50%	2	2/3	2/3	2/3				5.1				
*	50%	2	1/2	1/2	1/2				5.1				
*	25%	2	2/3	2/3	2/3				5.2		3.4	3.7	3.6
*	25%	2	1/2	1/2	1/2				5.2				
*	25%	.2	1	1/2	1/2			4.0					
*	50%	.2	1	1/2	1/2			3.2		4.1			
*	75%	.2	1	1/2	1/2			2.1		1.5			
*	25%	.5	1	1/2	1/2			4.1					
*	50%	.5	1	1/2	1/2			3.4					
*	75%	.5	1	1/2	1/2			2.7					
*	25%	1	1	1/2	1/2			4.6					
*	50%	1	1	1/2	1/2			3.9		3.8			
*	75%	1	1	1/2	1/2			3.4		2.1			
*	25%	2	1	1/2	1/2			5.1					
*	50%	2	1	1/2	1/2			4.7					
*	75%	2	1	1/2	1/2			4.4					
@	50%	.2	1	1/2	1/2			2.1		2.9			
@	75%	.2	1	1/2	1/2			1.3		2.2			
@	50%	1	1	1/2	1/2			2.8		3.2			
@	75%	1	1	1/2	1/2			2.4		2.0			
#	50%	.2	1	1/2	1/2			1.2		1.7			
#	75%	.2	1	1/2	1/2			.81		1.6			
#	50%	1	1	1/2	1/2			1.9		2.3			
#	75%	1	1	1/2	1/2			1.6		1.7			

* Multiple programming levels at the three site are 10/11/11.
@ Multiple programming levels at the three site are 16/8/8.
# Multiple programming levels at the three site are 24/4/4.
TZ : Average no. of requests per transaction (transaction size).
DZ : Total number of data items in the database (database size).
MP : Multiprogramming level.
R/(R+W) : Percentage of transactions that are read-only.
IO/Comm : Ratio of local delay to communication delay (excluding queueing delay).
No. of Copy : Fraction of the database residing at sites S1, S2, & S3.

Figure A.4 Average Response Per Write Request, Short Transactions & Communication Bound

TZ= 4, DZ= 8192

MP	R/(R+W)	IO / Comm	no. of copy S1	S2	S3	Cntrl Total	Basic Prmry
32	75%	.4/1/2	1	1/2	1/2	6.8/2.3	8.9/2.9
32	25%	.4/1/2	1	1/2	1/2	.93/2.7	.95/2.8
32	75%	2/1/2	1	1/2	1/2	5.6/1.9	6.5/2.1
32	25%	2/1 2	1	1/2	1/2	.91/2.7	.98/2.9
32	75%	.4/1/8	1	1/2	1/2	3.3/1.1	3.8/1.2
32	25%	.4/1/8	1	1/2	1/2	.39/1.2	.41/1.2
32	75%	2/1/8	1	1/2	1/2	3.1/1.1	3.6/1.2
32	25%	2/1/8	1	1/2	1/2	.40/1.2	
32	75%	.4/1/2	2/3	2/3	2/3	2.6/.87	
32	25%	.4/1/2	2/3	2/3	2/3	.66/1.9	
32	75%	2/1/2	2/3	2/3	2/3	2.6/.85	
32	25%	2/1 2	2/3	2/3	2/3	.75/1.9	
32	75%	.4/1/8	2/3	2/3	2/3		
32	25%	.4/1/8	2/3	2/3	2/3		
32	75%	2/1/8	2/3	2/3	2/3		
32	25%	2/1/8	2/3	2/3	2/3		
32	75%	.4/1/2	1	1/2	1/2	2.9/3.1	1.9/3.5
32	25%	.4/1/2	1	1/2	1/2	6.7/6.3	1.1/7.3
32	75%	2/1/2	1	1/2	1/2	4.0/4.1	3.5/3.6
32	25%	2/1 2	1	1/2	1/2	6.9/7.0	2.4/7.6
32	75%	.4/1/8	1	1/2	1/2	5.9/5.3	4.3/16
32	25%	.4/1/8	1	1/2	1/2	16/14	5.5/6.9
32	75%	2/1/8	1	1/2	1/2	6.8/6.7	
32	25%	2/1/8	1	1/2	1/2	14/14	
32	75%	.4/1/2	2/3	2/3	2/3	10/9	
32	25%	.4/1/2	2/3	2/3	2/3	11/12	
32	75%	2/1/2	2/3	2/3	2/3	10/9	
32	25%	2/1 2	2/3	2/3	2/3	11/12	
32	75%	.4/1/8	2/3	2/3	2/3		
32	25%	.4/1/8	2/3	2/3	2/3		
32	75%	2/1/8	2/3	2/3	2/3		
32	25%	2/1/8	2/3	2/3	2/3		

TZ = Average number of requests per transaction (transaction size).
DZ = Total number of data items in the database (database size).
MP = Multiprogramming level.
R/(R+W) = Percentage of transactions that are read-only.
IO/Comm = local delay/message communication delay/data communication delay
no. of copy = Fraction of the database residing at each site.

Figure A.5 Through-Put (Read/Write): Short Transactions & Communication Bound

TZ=4, MP=32, DZ=8192.

MP	R/W	IO/Com	Database Copies			Basic Basic	Prmry Prmry	Cntrl	Basic Prmry	Basic Cntrl	Basic Tstmp	Mltpl Versn	Basic Optms
*	.25	.2	1	1	1	.97/2.9	1.4/4.3	1.3/4.0	1.5/4.5		1.8/5.4	1.9/5	1.8/5.6
*	.50	.2	1	1	1	2.5/2.6	3.2/3.2	2.8/3.0					
*	.75	.2	1	1	1	5.6/1.8	5.5/1.8	5.0/1.6	7.1/2.5		8.6/2.9	8.1/2.7	8.1/2.7
*	.25	.5	1	1	1	.40/1.3	.63/1.8		.65/2.0				
*	.50	.5	1	1	1	1.1/1.1	1.4/1.4		1.6/1.6				
*	.75	.5	1	1	1	2.3/.79	2.4/.80	2.3/.76	3.1/1.0				
*	.25	1	1	1	1	.21/.64	.34/.94		.34/1.0				
*	.50	1	1	1	1	.58/.55	.72/.70		.81/.81				
*	.75	1	1	1	1	1.2/.39	1.2/.39	1.3/.39	1.6/.53				
*	.25	2	1	1	1	.11/.32					.21/.59	.21/.59	.22/.60
*	.50	2	1	1	1	.28/.28							
*	.75	2	1	1	1	.61/.20					.94/.31	.85/.28	.84/.28
*	.25	.2	2/3	2/3	2/3	1.2/3.6	1.5/4.7	1.3/3.9	1.5/4.7		1.7/4.9	2.0/6.0	2.0/5.9
*	.50	.2	2/3	2/3	2/3	2.9/2.9	3.3/3.3	2.9/2.8					
*	.75	.2	2/3	2/3	2/3	5.6/1.8	5.6/1.8	4.7/1.6	6.2/2.1		7.1/2.5	7.8/2.4	7.7/2.6
@	.75	.2	2/3	2/3	2/3			4.1/1.3					
#	.75	.2	2/3	2/3	2/3			3.4/1.1					
*	.25	.5	2/3	2/3	2/3	.53/1.6	.66/2.0		.68/2.0				
*	.50	.5	2/3	2/3	2/3	1.3/1.2	1.5/1.5		1.5/1.5				
*	.75	.5	2/3	2/3	2/3	2.3/.79	2.4/.80	2.2/.72	2.7/.84				
*	.25	1	2/3	2/3	2/3	.26/.81			.34/1.0				
*	.50	1	2/3	2/3	2/3	.63/.64			.80/.77				
*	.75	1	2/3	2/3	2/3	1.2/.39		1.1/.39	1.3/.46				
*	.25	1	2/3	2/3	2/3						.18/.54	.24/.72	.23/.71
*	.75	1	2/3	2/3	2/3						.78/.26	1.0/.32	.88/.29
*	.25	.2	1/2	1/2	1/2	1.4/4.3	1.6/4.9						
*	.50	.2	1/2	1/2	1/2	3.3/3.2	3.6/3.6						
*	.75	.2	1/2	1/2	1/2	5.7/1.9	5.8/1.8	4.6/1.5					
*	.25	.5	1/2	1/2	1/2	.60/1.8	.71/2.1		.66/2.2				
*	.50	.5	1/2	1/2	1/2	1.4/1.4	1.5/1.5		1.6/1.6				
*	.75	.5	1/2	1/2	1/2	2.5/.78	2.4/.88		2.8/.86				
*	.25	1	1/2	1/2	1/2	.32/.94	.37/1.1		.36/1.1				
*	.50	1	1/2	1/2	1/2	.72/.71	.80/.76		.81/.81				
*	.75	1	1/2	1/2	1/2	1.2/.43	1.3/.42		1.4/.47				
*	.25	.2	1	1/2	1/2			1.2/3.7					
*	.50	.2	1	1/2	1/2			2.8/2.7		3.1/3.2			
*	.75	.2	1	1/2	1/2			4.6/1.4		5.8/1.9			
*	.25	.5	1	1/2	1/2			.54/1.7					
*	.50	.5	1	1/2	1/2			1.2/1.2					
*	.75	.5	1	1/2	1/2			2.0/.67					
*	.25	1	1	1/2	1/2			.30/.85					
*	.50	1	1	1/2	1/2			.63/.63		.74/.68			
*	.75	1	1	1/2	1/2			1.1/.34		1.3/.41			
@	.50	.2	1	1/2	1/2			2.4/2.4		2.6/2.7			
@	.75	.2	1	1/2	1/2			4.0/1.3		5.2/1.6			
@	.50	1	1	1/2	1/2			.54/.54		.62/.60			
@	.75	1	1	1/2	1/2			.90/.31		1.1/.38			
#	.50	.2	1	1/2	1/2			2.0/2.1		2.2/2.2			
#	.75	.2	1	1/2	1/2			3.4/1.1		4.0/1.3			
#	.50	1	1	1/2	1/2			.45/.44		.48/.49			
#	.75	1	1	1/2	1/2			.75/.25		.88/.28			

* Multiple programming levels at the three site are 10/11/11.
@ Multiple programming levels at the three site are 16/8/8.
# Multiple programming levels at the three site are 24/4/4.

Assumptions:
Queueing for local processing is simulated.
Two kinds of local processing delay are simulated:
message processing delay and data processing delay.
The average round trip communication is fixed at 1
The message processing delay is fixed at 5% of the
5% of round trip communication delay
Ratio of data processing & message processing delay is 10
The ratio of data processing delay to round trip
communication delay is shown in column 'IO/Com'

Notation:
TZ = Average number of requests per transaction (transaction size).
DZ = Total number of data items in the database (database size).
MP = Multiplr programming level.
R/W = Percentage of transactins that are read-only.
IO/Com = Ratio of local data processing delay to communication
delay (excluding queueing).
Database Copies = Fraction of the database residing at each site.

Figure A.6 Through-Put (Read/Write), Short
Transacticns & IO Bounded

TZ=4, MP= 32, DZ= 8192

MP	R/W	IO/Com	Database Copies			Basic Basic	Prmry Prmry	Cntrl Total	Basic Prmry	Basic Cntrl	Basic Tstmp	Mltpl Versn	Basic Optms
*	.25	.2	1	1	1	3.6/8.7	4.0/5.4	5.5/5.7	2.3/5.5		3.0/3.0	3.4/3.5	3.4/3.5
*	.50	.2	1	1	1	3.3/8.1	4.0/5.4	5.3/5.4					
*	.75	.2	1	1	1	2.9/7.3	4.0/5.3	5.0/5.1	2.2/5.6		2.3/2.3	2.9/3.0	2.9/3.1
*	.25	.5	1	1	1	9.0/20	9.2/12		5.5/12				
*	.50	.5	1	1	1	8.0/18	9.1/12		5.4/12				
*	.75	.5	1	1	1	7.1/16	9.0/12	11/11	5.4/13				
*	.25	1	1	1	1	17/38	18/24		11/24				
*	.50	1	1	1	1	16/35	18/24		11/24				
*	.75	1	1	1	1	14/31	18/24	20/20	11/24				
*	.25	2	1	1	1	35/76					27/27	33/33	33/33
*	.50	2	1	1	1	32/70							
*	.75	2	1	1	1	28/61					22/22	29/29	29/29
*	.25	.2	2/3	2/3	2/3	3.8/6.6	4.6/4.7	5.8/6.0	3.2/4.9		3.7/3.6	2.9/3.3	3.4/3.5
*	.50	.2	2/3	2/3	2/3	3.7/6.4	4.4/4.6	5.5/5.7					
*	.75	.2	2/3	2/3	2/3	3.5/6.2	4.3/4.4	5.4/5.5	3.2/5.0		3.0/3.0	2.4/2.7	3.2/3.4
@	.75	.2	2/3	2/3	2/3			6.1/6.2					
#	.75	.2	2/3	2/3	2/3			7.1/7.3					
*	.25	.5	2/3	2/3	2/3	9.2/15	11/11		7.4/11				
*	.50	.5	2/3	2/3	2/3	8.8/15	10/10		7.5/11				
*	.75	.5	2/3	2/3	2/3	8.4/14	9.8/9.9	11/12	7.6/11				
*	.25	1	2/3	2/3	2/3	18/29			15/22				
*	.50	1	2/3	2/3	2/3	17/28			15/22				
*	.75	1	2/3	2/3	2/3	16/27		22/22	15/22				
*	.25	1	2/3	2/3	2/3						33/34	26/28	30/30
*	.75	1	2/3	2/3	2/3						12/12	20/21	29/29
*	.25	.2	1/2	1/2	1/2	4.0/5.4	4.2/4.5						
*	.50	.2	1/2	1/2	1/2	3.8/5.2	4.2/4.4						
*	.75	.2	1/2	1/2	1/2	3.7/5.1	4.1/4.3	5.5/5.7					
*	.25	.5	1/2	1/2	1/2	9.2/12	9.9/10		7.4/10				
*	.50	.5	1/2	1/2	1/2	9.1/12	9.7/9.9		7.5/10				
*	.75	.5	1/2	1/2	1/2	8.8/11	9.4/9.6		7.6/11				
*	.25	1	1/2	1/2	1/2	18/24	19/20		15/20				
*	.50	1	1/2	1/2	1/2	18/23	19/19		15/20				
*	.75	1	1/2	1/2	1/2	17/22	18/19		15/21				
*	.25	.2	1	1/2	1/2			6.0/6.1					
*	.50	.2	1	1/2	1/2			5.7/5.9		3.1/6.4			
*	.75	.2	1	1/2	1/2			5.5/5.6		3.2/6.7			
*	.25	.5	1	1/2	1/2			13/14					
*	.50	.5	1	1/2	1/2			13/13					
*	.75	.5	1	1/2	1/2			12/12					
*	.25	1	1	1/2	1/2			26/26					
*	.50	1	1	1/2	1/2			25/25		14/29			
*	.75	1	1	1/2	1/2			23/24		14/30			
@	.50	.2	1	1/2	1/2			6.3/6.4		4.0/6.9			
@	.75	.2	1	1/2	1/2			6.0/6.2		3.7/7.0			
@	.50	1	1	1/2	1/2			28/28		17/31			
@	.75	1	1	1/2	1/2			27/27		17/32			
#	.50	.2	1	1/2	1/2			7.0/7.2		5.4/7.4			
#	.75	.2	1	1/2	1/2			6.7/6.8		5.0/7.3			
#	.50	1	1	1/2	1/2			32/33		25/34			
#	.75	1	1	1/2	1/2			31/31		23/34			

* Multiple programming levels at the three site are 10/11/11.
@ Multiple programming levels at the three site are 16/8/8.
# Multiple programming levels at the three site are 24/4/4.

Assumptions:
Queueing for local processing is simulated.
Two kinds of local processing delay are simulated:
message processing delay and data processing delay.
The average round trip communication is fixed at 1
The message processing delay is fixed at 5% of the
5% of round trip communication delay
Ratio of data processing & message processing delay is 10
The ratio of data processing delay to round trip
communication delay is shown in colume 'IO/Com'

Notation:
TZ = Average number of requests per transaction (transaction size).
DZ = Total number of data items in the database (database size).
MP = Multiplr programming level.
R/W = Percentage of transactins that are read-only.
IO/Com = Ratio of local data processing delay to communication
delay (excluding queueing).
Database Copies = Fraction of the database residing at each site.

Figure A.7 Average Response Time (Read/Write):
Short Transactions & IO Bound

TZ=16, DZ=8192, MP=32

MP	R/W	IO/ Com	Database Copies			Basic Prmry	Basic Tstmp	Mltpl Versn	Basic Optms
*	.25	.2	1	1	1	2.0/6.0	1.5/4.4	.90/.20	.85/1.9
*	.75	.2	1	1	1	9.2/3.0	7.9/2.8	6.6/2.1	6.6/2.4
*	.25	2	1	1	1	.25/.83	.16/.46	.09/.20	.09/.20
*	.75	2	1	1	1	1.1/.37	.90/.28	.69/.22	.95/.29
*	.25	.2	2/3	2/3	2/3	1.8/5.6	1.4/3.9	2.1/6.1	2.1/6.2
*	.75	.2	2/3	2/3	2/3	7.9/2.7	6.7/1.9	10./3.5	9.6/3.4
*	.25	2	2/3	2/3	2/3	.25/.81	.14/.39	.26/.78	.23/.85
*	.75	2	2/3	2/3	2/3	.94/.33	.68/.22	1.4/.44	1.3/.37

* Multiple programming levels at the three site are 10/11/11.
Ratio of local data processing & message processing delay is 10

Assumption:
Queueing for local processing is simulated.
Two kinds of local processing are simulated:
(message and data processing).
The round trip communication is fixed at 1
The local message processing delay is fixed at
5% of the round trip communication delay
The ratio of local data processing delay to round trip
communication delay is shown in colume 'IO/Comm'

Notation:
TZ = Average number of requests per transaction.
DZ = Total number of data items in the database.
MP = Multiple programming level.
R/W = Ratio of read-only to write transactions.
IO/Com = Ratio of local data processing delay to
communication delay (excluding queueing).
Database Copies = Fraction of the database at each site.

Figure A.8 Through-Put (Read/Write): Long Transaction Loaded & IO Bound

TZ=16,DZ=8192,MP=32

MP	R/W	IO/ Com	Database Copies			Basic Prmry	Basic Tstmp	Mltpl Versn	Basic Optms
*	.25	.2	1	1	1	2.8/4.6	2.2/2.2	1.1/2.7	2.1/2.6
*	.75	.2	1	1	1	1.9/5.5	2.2/2.2	1.6/2.8	1.7/3.0
*	.25	2	1	1	1	20./33	22./22	13/25	19/24
*	.75	2	1	1	1	17./42	21/21	19/26	21/27
*	.25	.2	2/3	2/3	2/3	3.3/4.8	3.0/3.0	1.4/2.6	2.3/2.9
*	.75	.2	2/3	2/3	2/3	2.4/5.6	2.9/2.9	1.9/2.6	2.0/3.4
*	.25	1	2/3	2/3	2/3	25./33	29/29	14/19	18/21
*	.75	1	2/3	2/3	2/3	22./42	28/28	16/20	18/25

* Multiple programming levels at the three site are 10/11/11.
Ratio of local data processing & message processing delay is 10

Assumption:
Queueing for local processing is simulated.
Two kinds of local processing are simulated:
(message and data processing).
The round trip communication is fixed at 1
The local message processing delay is fixed at
5% of the round trip communication delay
The ratio of local data processing delay to round trip
communication delay is shown in colume 'IO/Comm'

Notation:
TZ = Average number of requests per transaction.
DZ = Total number of data items in the database.
MP = Multiple programming level.
R/W = Ratio of read-only to write transactions.
IO/Com = Ratio of local data processing delay to
communication delay (excluding queueing).
Database Copies = Fraction of the database at each site.

Figure A.9 Average Response Time: Long
Transaction Loaded & IO Bound

TZ=16,DZ=8192,MP=32

MP	R/W	IO/Com	Database Copies			Basic Prmry	Basic Tstmp	Mltpl Versn	Basic Optms
*	.25	.2	1	1	1	2.4/7.3	9/26	3.6/8.6	4.3/10
*	.75	.2	1	1	1	21 /6.5	64/19	38/11	40/12
*	.25	2	1	1	1	1.2/3.5	1.0/2.8	.42/.98	.46/1.3
*	.75	2	1	1	1	5.4/1.8	8.0/2.2	10/2.6	6.2/1.9
*	.25	.2	2/3	2/3	2/3	2.2/6.5	1.6/4.4	.97/2.7	1.9/5.1
*	.75	.2	2/3	2/3	2/3	9.9/3.2	7.9/2.4	7.5/2.8	10/3
*	.25	2	2/3	2/3	2/3	1.0/2.9	.84/2.4	.54/1.5	.66/2.0
*	.75	2	2/3	2/3	2/3	4.7/1.5	4.0/1.3	5.6/1.7	4.8/1.3

* Multiple programming levels at the three site are 10/11/11.

Assumption:
Queueing for communication channel is simulated.
Only one kind of local processing is simulated.
The average round trip communication is fixed at 1
The ratio of local data processing delay to round trip
communication delay is shown in colume 'IO/Comm'

Notation:
TZ = Average number of requests per transaction.
DZ = Total number of data items in the database.
MP = Multiple programming level.
R/W = Rati of read-only to write transactions.
IO/Com = Ratio of local processing delay to communication
delay (excluding queueing delay).
Database Copies = Fraction of the database at each site.

Figure A.10 Through-Put (Read/Write): Long Transaction
Loaded & Communicaton Bound

MP	R/W	IO/Com	Database Copies			Basic Prmry	Basic Tstmp	Mltpl Versn	Basic Optms
*	.25	.2	1	1	1	1.2/4.1	.2/.2	.2/.53	.42/.51
*	.75	.2	1	1	1	.52/3.1	.2/.2	.2/.45	.30/.50
*	.25	2	1	1	1	3.9/7.8	2/2	2/4.9	3.5/4.6
*	.75	2	1	1	1	3.1/8.8	2/2	2/3.1	2.8/4.3
*	.25	.2	2/3	2/3	2/3	2.5/4.2	2/2	.86/3.7	2.2/2.8
*	.75	.2	2/3	2/3	2/3	2.1/4.2	1.9/1.9	1.4/3.3	1.9/3.1
*	.25	1	2/3	2/3	2/3	6.3/8.9	3.1/3.2	3.0/7.8	5.6/6.7
*	.75	1	2/3	2/3	2/3	4.2/8.5	3.2/3.1	3.1/5.7	4.2/6.6

* Multiple programming levels at the three site are 10/11/11.
Assumption:
Queueing for communication channel is simulated.
Only one kind of local processing is simulated.
The average round trip communication is fixed at 1
The ratio of local data processing delay to round trip
communication delay is shown in colume 'IO/Comm'

Notation:
TZ = Average number of requests per transaction.
DZ = Total number of data items in the database.
MP = Multiple programming level.
R/W = Rati of read-only to write transactions.
IO/Com = Ratio of local processing delay to communication
delay (excluding queueing delay).
Databae Copies = Fraction of the database at each site.

Figure A.11 Average Response Time (Read/Write)
Long Transaction & Communication Bound

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.


DAT
ILM